ALEXANDRE RAFAEL LENZ

UTILIZANDO TÉCNICAS DE APRENDIZADO DE MÁQUINA PARA APOIAR O TESTE DE REGRESSÃO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientadora: Profa. Dra. Silvia Regina Vergilio
Co-Orientadora: Profa. Dra. Aurora Pozo

CURITIBA

2009

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Alexandre Rafael Lenz, avaliamos o trabalho intitulado, *"UTILIZANDO TÉCNICAS DE APRENDIZADO DE MÁQUINA PARA APOIAR O TESTE DE REGRESSÃO"*, cuja defesa foi realizada no dia 31 de agosto de 2009, às 9:00 horas, no Auditório do Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 31 de agosto de 2009.

Profª. Dra. Silvia Regina Vergilio
DINF/UFPR – Orientadora

Profª. Dra. Aurora Trinidad Ramirez Pozo
DINF/UFPR – Coorientadora

Profª. Dra. Elisa Hatsue Moriya Huzita
UEM – Membro Externo

Prof. Dr. Eduardo Spinosa
DINF/UFPR – Membro Interno

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

AM - Aprendizado de Máquina

API - *Application Program Interface*

C4, C4.5, C5.0 e J48 - *Decision Trees*

C-USO - Uso Computacional

CART - *Classification and Regression Trees*

CHAID - *Chi Square Automatic Interaction Detection*

COBWEB - Algoritmo de agrupamento hierárquico conceitual

EM - *Expectation Maximization*, Maximização das Estimativas

FCFD - Família de Critérios de Fluxo de Dados

FCPU - Família de Critérios Potenciais Usos

IEEE - *Institute of Electrical and Electronics Engineers*, Instituto de Engenheiros Eletricistas e Eletrônicos

IFN - Rede *Info-Fuzzy*

K-MEANS - Algoritmo das k-médias

KDD - *Knowledge Discovery in Databases*, Descoberta de Conhecimento em Bancos de Dados

MD - Mineração de Dados

P-USO - Uso Predicativo

PART - Algoritmo que produz um conjunto de regras a partir de uma árvore gerada pelo algoritmo C4.5.

PDG - *Program Dependence Graph*

PDU - Todos-potenciais-du-caminhos

PG - Progressão Geométrica

PROTEUM - *Program Testing Using Mutants*

POKE-TOOL - *Potential-Uses Criteria Tool for Program Testing*

PU - Todos-potenciais-usos

PUDU - Todos-potenciais-usos/DU

RITA - *Relating Information from Testing Activity*

SDG - *System Dependence Graph*

TDIDT - ID3 - *Top-Down Induction of Decision Trees*

WEKA - *Waikato Enviroment for Knowledge Analysis*

ZEROR - Algoritmo simples que calcula o atributo mais preditivo e classifica os objetos usando apenas este atributo. Utiliza previsão da média ou da moda.

RESUMO

Independentemente do tipo de manutenção realizada, o teste de regressão é indispensável para testar as modificações e as novas funcionalidades do software. Ele também é responsável por verificar se as funcionalidades existentes não foram negativamente afetadas pela modificação. Muitas técnicas têm sido propostas para reduzir os esforços e aumentar a eficácia dos testes de regressão. Dentre elas, algumas utilizando Aprendizado de Máquina (AM). Entretanto, a maioria dos trabalhos existentes não relacionam as informações coletadas durante o teste provenientes da aplicação de diferentes técnicas e critérios de teste. Esses critérios são considerados complementares porque podem revelar diferentes tipos de defeitos, e considerar essa complementariedade pode auxiliar o teste de regressão, reduzindo os esforços gastos nesta atividade. Dada essa perspectiva, este trabalho tem como objetivo explorar técnicas de AM, como de agrupamento, para relacionar informações como, por exemplo: dados de entrada, saída produzida, elementos cobertos por critérios estruturais, defeitos revelados, e etc. Com estas informações os dados são agrupados em classes funcionais. Os resultados assim obtidos são então submetidos a um algoritmo de classificação, para geração de regras a serem utilizadas na seleção e priorização de dados de teste. Uma ferramenta, chamada RITA (*Relating Information from Testing Activity*), foi implementada para dar suporte à abordagem proposta. Ela foi utilizada em experimentos, cujos resultados mostram a aplicabilidade da abordagem e uma redução de custo do teste de regressão.

ABSTRACT

Regression testing activities are necessary to test the modifications and the eventual new features of the modified software. They are also required to check whether the existent features were not adversely affected by the introduced modifications. Many works have proposed techniques to reduce the efforts and to increase the effectiveness of the regression testing activities. Some of them are based on Machine Learning. However, most of the existent works do not relate information from the application of different test techniques and criteria. These criteria are considered complementary because they can reveal different kind of faults, and to use this characteristic can reduce the effort spent in the regression testing activities. Considering this fact, this work explores the use of Machine Learning techiniques in the regression testing. An approach is introduced, that uses clustering techniques to relate test information like: inputs, produced output, coverage elements required by structural criteria, revealed faults, and etc. With this information, the input data are grouped in functional classes and the obtained results are submitted to classifiers to generate rules to be used for selection and prioritization of teste data. A tool, named RITA (Relating Information from Testing Activity) was implemented to support the approach. By using RITA an evaluation experiment was conducted and the obtained results show the applicability of the introduced approach and that it contributes to reduce the regression test costs.

CAPÍTULO 1

INTRODUÇÃO

1.1 Contexto

A Engenharia de Software tem como objetivo introduzir novos métodos, ferramentas e modelos de desenvolvimento de software e isso tem sido efetuado em um ritmo crescente. Esse fato é impulsionado pela necessidade de se produzir software com mais velocidade e qualidade, o que, por sua vez, desencadeou um processo de melhoria contínua que perdura até hoje. Entende-se por sistemas com qualidade, aqueles que atendem aos requisitos de funcionalidade, eficiência, portabilidade, usabilidade e confiabilidade [24]. No entanto, a construção de software com estes requisitos não é uma tarefa trivial.

A satisfação do usuário é o propósito da qualidade de um produto. Apesar de parecer simples, o objetivo da qualidade no que diz respeito a software é um domínio bastante obscuro. Normalmente são fatores muito difíceis de medir que têm maior peso para o usuário [48]. Contudo, os fatores relacionados à quantidade de defeitos revelados e à confiabilidade do software são considerados muito importantes. Para tanto, os estudos sobre teste de software estão sob constante evolução, tendo como objetivo a proposição de novas abordagens e técnicas, assim como o aprimoramento das existentes.

Sendo assim, uma das principais atividades de garantia da qualidade de um produto de software é o teste. O teste de software é uma das áreas de pesquisa da Engenharia de Software e constitui um dos elementos para aprimorar a produtividade e fornecer evidências da confiabilidade do software. O objetivo principal do teste de software é revelar o número máximo de defeitos dispondo do mínimo de esforço. As atividades de teste envolvem basicamente quatro etapas: planejamento de testes, projeto de casos de teste, execução e avaliação dos resultados [5, 64, 70]. Para tanto utiliza-se um conjunto de técnicas e critérios, teoricamente fundamentados, que sistematizam essas atividades.

Essas atividades são realizadas em quatro diferentes níveis: nível de unidade, nível

de integração, nível de validação e nível de sistema. Geralmente, os critérios de teste de software são estabelecidos a partir de três técnicas: funcional, estrutural e baseada em defeitos. Cada uma dessas técnicas estabelece os requisitos de teste a partir de diferentes aspectos do software. A técnica funcional estabelece os requisitos de teste a partir da especificação do programa. A técnica estrutural estabelece os requisitos a partir da estrutura do código-fonte do programa, enquanto que a técnica baseada em defeitos estabelece os requisitos de teste a partir de defeitos típicos cometidos no processo de desenvolvimento de software. Essas técnicas são vistas como complementares, pois revelam diferentes tipos de defeitos.

As fases de teste e manutenção consomem aproximadamente 50% dos recursos utilizados no desenvolvimento de sistemas [70]. Por esse motivo, essas etapas estão sendo consideradas como muito importantes no ciclo de desenvolvimento. Os principais motivos para tal concentração de recursos são: o caráter destrutivo da atividade; sistemas mal especificados nos quais há omissão e especificação errônea de requisitos; a rara utilização sistemática de técnicas de teste e manutenção em comparação às utilizadas nas primeiras fases do desenvolvimento; o caráter evolutivo dos sistemas de software devido à adição de funcionalidades.

Considerando o seu caráter evolutivo, o software pode sofrer modificações por várias razões como correções de defeitos não encontrados na fase de desenvolvimento, mudanças de especificação, adaptação a um novo ambiente, etc. Essas modificações podem fazer com que partes do software, que antes funcionavam perfeitamente, parem de funcionar. Isto refere-se ao problema da introdução de novos defeitos, para o qual a realização de teste de regressão é fundamental. O teste de regressão é uma atividade de suma importância, visto que garante que o caráter evolutivo do software não acarrete a introdução de defeitos em partes do sistema que antes funcionavam corretamente.

Dada a impossibilidade de reexecução de todos os casos de teste durante o teste de regressão, visto que esta atividade demanda muito tempo e esforço, têm-se os problemas da seleção, priorização e redução de um conjunto de casos de teste. Estes problemas foram amplamente explorados e avaliados [77], essas abordagens utilizam diferentes técnicas,

dentre estas técnicas de Aprendizado de Máquina (AM) [9, 11, 52, 56, 59, 85, 96].

As técnicas de AM são amplamente utilizadas na resolução de diversos problemas do cotidiano [12, 62]. Como exemplos desses problemas têm-se: a) veículos autônomos que aprendem a dirigir em vias expressas; b) reconhecimento da fala; c) detecção de fraudes em cartões de crédito; d) estratégias para a construção de jogos; e) sistemas biométricos; f) sistemas financeiros; e g) mineração de dados para descobrir regras gerais em bases de dados.

O último problema citado encaixa-se no contexto da melhoria de um conjunto de casos de teste de regressão, onde as técnicas de AM podem ser muito úteis. Elas podem ser utilizadas como um oráculo, o qual é capaz de gerar regras que são intuitivas e de simples entendimento e estas, por sua vez, são utilizadas como base para a seleção dos testes de regressão. Como exemplos de técnicas de AM têm-se: algoritmos de agrupamento, redes neurais, árvores de decisão, redes *bayesianas*, entre outras.

Cada abordagem proposta para seleção de casos de teste de regressão focaliza uma determinada técnica de teste. Como exemplos têm-se: a abordagem baseada em equações lineares que foca o teste estrutural [29], a abordagem baseada em execução simbólica que foca o teste funcional [95] e a abordagem baseada em mutação seletiva que foca o teste baseado em defeitos [61]. Além disso, como citado anteriormente, algumas abordagens foram propostas com o intuito de utilizar técnicas de (AM) para seleção e priorização de casos de teste [52, 56, 85, 89]. Redes Neurais e *Info-Fuzzy Networks* já foram utilizadas em estudos anteriores obtendo bons resultados na seleção de casos de teste de regressão, sendo que nesses estudos foi utilizada apenas a técnica de teste funcional como base para seleção dos casos de teste.

As abordagens citadas anteriormente, contudo, não são integradas com ferramentas que implementam critérios de teste, não exploram o relacionamento entre características e dados de execução de diferentes técnicas de teste e nem a complementariedade das técnicas em revelar diferentes tipos de defeitos. É importante considerar a complementariedade das técnicas em revelar diferentes tipos de defeitos, visto que a análise das informações de execução de testes, no contexto do teste de regressão, seria inviável por um usuário,

principalmente para programas grandes. A utilização de diferentes ferramentas e critérios possibilita a geração de diferentes informações, sendo que as técnicas de AM podem reduzir custo e esforço na obtenção automática de relações entre essas informações.

1.2 Motivação

Dado o contexto que reflete o estado da arte atual em relação à utilização de AM para apoio ao teste de regressão, têm-se os seguintes pontos, que justificam o presente trabalho:

- A necessidade de se produzir software com qualidade;
- A importância do teste de software como atividade de garantia da qualidade;
- A importância de se realizar testes de regressão durante a manutenção de sistemas, para garantir que as modificações não insiram defeitos em partes não modificadas;
- A necessidade da realização de testes de regressão contando com um conjunto de casos de teste compacto, para redução do tempo de execução e esforço sem perder qualidade;
- O fato de que as diferentes técnicas de teste podem ser aplicadas de forma complementar, revelando classes diferentes de defeitos, o que geralmente não é considerado pelas técnicas utilizadas no teste de regressão;
- As técnicas de AM têm sido utilizadas com resultados promissores na atividade de teste de regressão, mas as abordagens existentes consideram apenas um tipo de técnica de teste e a maioria delas não está integrada a uma ferramenta de teste;
- A ausência de uma ferramenta que automatize e melhore a seleção de casos de teste para o teste de regressão, a partir do relacionamento das características de diferentes técnicas de teste.

1.3 Objetivos

Este trabalho introduz uma abordagem genérica para apoiar o teste de regressão baseada em técnicas de Aprendizado de Máquina. As informações produzidas durante a atividade de teste, oriundas das técnicas estrutural e baseada em defeitos, são relacionadas e, por meio de uma técnica de agrupamento, são identificadas as classes de equivalência do programa em teste. Em seguida, as informações das três técnicas de teste (estrutural, funcional e baseada em defeitos), servem como entrada para uma técnica de classificação. Essa técnica de classificação gera regras que são aplicadas ao teste de regressão apoiando a priorização, redução e seleção de casos de teste.

Para apoiar a abordagem proposta, uma ferramenta é de suma importância. Sendo assim, foi desenvolvida a ferramenta RITA (***R****elating* ***I****nformation from* ***T****esting* ***A****ctivity*) que possibilita a execução de um conjunto de casos de teste sobre critérios de teste estrutural e baseado em defeitos. Para cobrir tais critérios, a abordagem é implementada, respectivamente, pelas ferramentas *Poke-Tool* e *Proteum*.

1.4 Organização do Trabalho

Este capítulo apresentou o contexto no qual o trabalho está inserido, a motivação para realizá-lo e os objetivos a serem atingidos. Os Capítulos 2, 3 e 4 apresentam uma revisão bibliográfica pertinente aos principais conceitos relacionados a este trabalho. O Capítulo 2 explora os conceitos relacionados à Inteligência Artificial e às técnicas de AM, enquanto o Capítulo 3 enfatiza os aspectos das técnicas e critérios de teste de software. Já o Capítulo 4 foca os conceitos referentes ao teste de regressão. O Capítulo 5 apresenta a abordagem genérica proposta e a ferramenta desenvolvida para apoiar o uso de tal abordagem, demonstrando de forma detalhada um exemplo de utilização. O Capítulo 6 apresenta os resultados dos experimentos de validação da abordagem proposta através de sua aplicação utilizando a ferramenta RITA em quatro programas. Por fim, o Capítulo 7 apresenta a conclusão e os trabalhos futuros. O trabalho também contém um apêndice (Apêndice A) que apresenta operadores de mutação da ferramenta *Proteum*.

CAPÍTULO 2

APRENDIZADO DE MÁQUINA

Este capítulo apresenta os principais conceitos relacionados ao Aprendizado de Máquina (AM). Inicialmente, são introduzidos os conceitos de descoberta de conhecimento e mineração de dados para possibilitar o entendimento das tarefas e técnicas de mineração de dados utilizadas neste trabalho. Em seguida, cada uma das técnicas utilizadas são detalhadas e a ferramenta *Weka* é apresentada sucintamente.

2.1 Terminologia e Conceitos Básicos

Esta seção apresenta, de forma resumida, as principais terminologias e conceitos utilizados na descoberta de conhecimento utilizando mineração de dados para construção de modelos, extração automática de padrões e exploração visual de dados.

2.1.1 Descoberta de Conhecimento

A descoberta de conhecimento em bancos de dados (*Knowledge Discovery in Databases - KDD*) é uma tecnologia que possui técnicas poderosas para a descoberta eficiente de conhecimento em uma grande coleção de dados, visando o auxílio no suporte à decisão. Segundo Fayyad et al. [28], "KDD é o processo não trivial de identificação, a partir de dados, de padrões que sejam válidos, novos, potencialmente úteis e compreensíveis".

Na definição de Fayyad, KDD é descrito como um processo geral de descoberta de conhecimento composto por várias etapas, incluindo: preparação dos dados, mineração de informação e avaliação do conhecimento. O termo não trivial significa que envolve algum mecanismo de busca ou inferência. Os padrões descobertos devem ser válidos diante de novos dados com algum grau de certeza. Estes padrões podem ser considerados conhecimento, dependendo de sua natureza. Os padrões devem ser novos, compreensíveis e úteis, ou seja,deverão trazer algum benefício novo que possa ser compreendido rapidamente pelo

usuário para tomada de decisão.

Carvalho [13] comenta que KDD é uma área interdisciplinar específica que surgiu em resposta à necessidade de novas abordagens e soluções para viabilizar a análise de grandes bancos de dados. O termo KDD é empregado para descrever todo o processo de extração de conhecimento de um conjunto de dados, enquanto que o termo mineração de dados (MD), refere-se a uma das etapas deste processo. A relação existente entre KDD e MD pode ser visualizada graficamente através da Figura 2.2.

Figura 2.1: Interligação entre KDD e Mineração de Dados [13]

De acordo com Rezende et al. [75], mineração de dados é a interação entre o especialista do domínio, o analista e o usuário final. O especialista do domínio deve possuir amplo conhecimento sobre o assunto da aplicação e deve fornecer apoio para a execução do processo. O analista é o especialista no processo de extração do conhecimento e responsável por sua execução. Ele deve conhecer profundamente as etapas que compõem o processo. O usuário final representa aqueles que utilizam o conhecimento extraído no processo para auxiliá-lo em um processo de tomada de decisão.

O processo de mineração de dados envolve várias etapas complexas, que devem ser executadas corretamente, pois cada etapa é fundamental para que os objetivos estabelecidos e o sucesso completo da aplicação sejam alcançados. O processo de mineração é tanto iterativo quanto interativo. A iteratividade tem sua natureza justificada pelo fato de que o conhecimento descoberto apresentado ao analista pode ser usado da seguinte forma: como base para a medida de avaliação a ser aprimorada; como base para a mineração a ser refinada; novos dados podem ser selecionados ou transformados; ou, ainda, novas fontes de dados podem ser integradas para adquirir resultados diferentes e mais apropriados. Portanto, o processo pode ser realizado em etapas seqüenciais de maneira que seja possível sua volta às etapas anteriores, criando laços de ligação entre elas. O

analista é, também, o responsável pela tomada de várias decisões, como na modelagem das informações, o tipo de algoritmo a ser usado e quais objetivos serão seguidos na busca do conhecimento, garantindo-se assim a sua natureza interativa.

Fayyad et al. [28] apresentam um processo típico de extração de conhecimento em base de dados, como é mostrado na Figura 2.2.

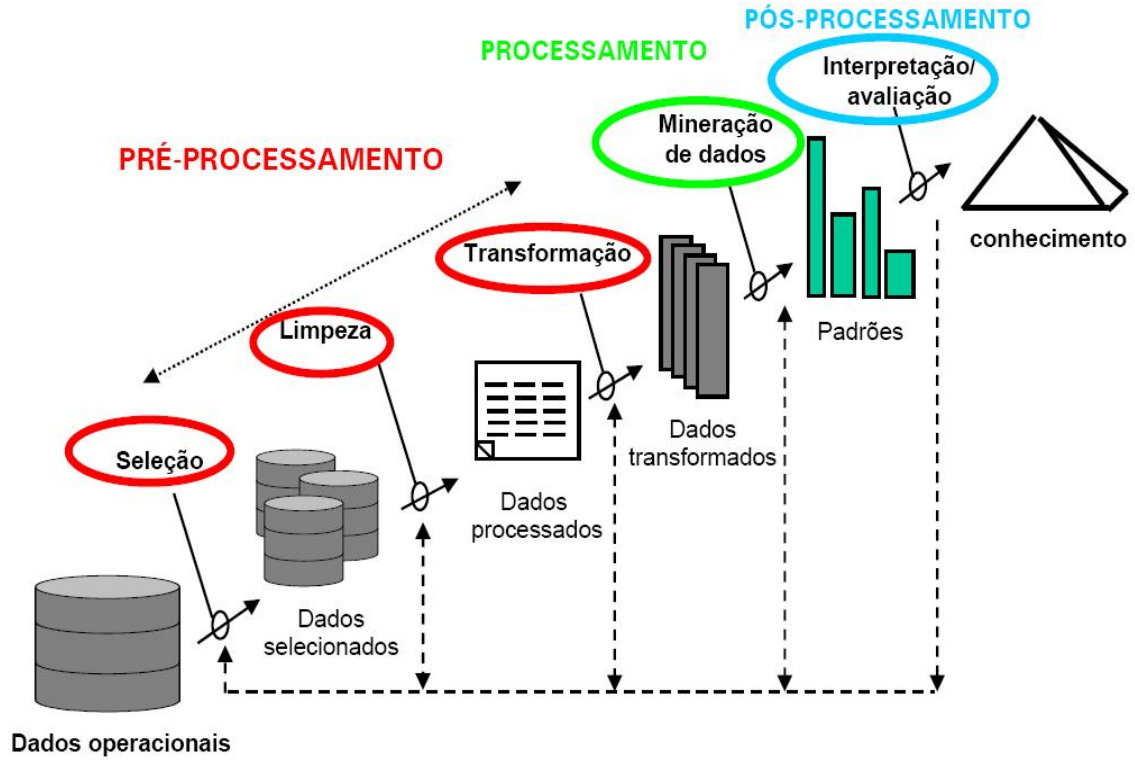


Figura 2.2: Visão geral das etapas que compõem o processo de descoberta de conhecimento. [28]

O processo de descoberta de conhecimento possui três passos. Inicialmente é preciso selecionar os tipos de dados que serão usados pelo algoritmo de mineração. Dados crus geralmente são variados, não estão organizados e nem todos são necessários para a mineração. Um grande esforço é necessário para se coletar uma boa quantidade de dados e transportá-los para um lugar onde se possa minerá-los. O primeiro passo é pré-processar os dados para aprontá-los para a análise. Usualmente os dados têm que ser formatados, amostrados, adaptados e, algumas vezes, transformados para que possam ser usados pelo algoritmo de mineração. Ocorre, então, o desenvolvimento do entendimento do domínio da aplicação, avaliação do hardware e software disponíveis, seleção, limpeza e transformação dos dados.

Após o pré-processamento, os dados estão prontos para serem minerados por um algoritmo. É a fase conhecida como a própria mineração de dados. É definida a escolha da tarefa e das técnicas a serem utilizadas, identificação da ferramenta que satisfaça a essas condições e aplicação desta aos dados. Este passo pode envolver técnicas muito diversas e a informação descoberta é usada, principalmente, para construção de modelos, extração automática de padrões e exploração visual de dados. O último passo do processo de mineração de dados é assimilar a informação minerada, chamado pós-processamento. É a interpretação dos resultados e a incorporação do conhecimento adquirido. No caso da construção de modelos, este passo consiste em avaliar a robustez e efetividade dos modelos produzidos. No caso da extração de padrões e exploração visual de dados, este passo consiste em tentar interpretar a informação extraída.

2.1.1.1 Caracterização e Representação dos Dados

Um conjunto de dados é uma coleção de objetos e seus atributos. Um atributo é uma propriedade ou característica de um objeto e também pode ser conhecido como variável, campo ou parâmetro. Uma coleção de atributos descreve um objeto. Objeto é também conhecido como registro, observação, ponto, entidade ou instância.

Normalmente, existem dois tipos de atributos: nominal, quando não existe uma ordem entre os valores e contínuo, quando existe uma ordem linear nos valores. Independente do tipo do atributo, um conjunto de atributos unido com um conjunto de instâncias representa o conjunto de dados. Diversos tipos de conjunto de dados podem existir a fim de representar o conjunto de dados. Entre eles, existem conjunto de dados baseados em registros, como matriz de dados, coleção de documentos e dados transacionais; conjunto de dados baseados em gráficos; baseados em uma ordem (seqüência), como dado espacial, dado temporal, dado seqüencial e dado com seqüência genética.

2.1.1.2 Tarefas de Mineração de Dados

Para utilização de mineração de dados, é preciso definir a escolha da tarefa e das técnicas a serem utilizadas, identificar a ferramenta que atinja os objetivos propostos e aplicar esta

ferramenta aos dados. Este passo pode envolver técnicas muito diversas.

É importante distinguir o que é uma tarefa e o que é uma técnica de mineração. A tarefa consiste na especificação do que está querendo buscar nos dados, que tipo de regularidades ou categoria de padrões tem interesse em encontrar, ou que tipo de padrões poderia surpreender. A técnica de mineração consiste na especificação de métodos que garantam como descobrir os padrões que interessam.

As tarefas de mineração podem ser classificadas em duas grandes áreas: predição e descrição. Dentre essas áreas, dependendo do objetivo procurado, dividem-se as principais categorias. São elas: classificação, regressão, associação e agrupamento. A Figura 2.3 mostra um resumo das principais tarefas de mineração de dados.

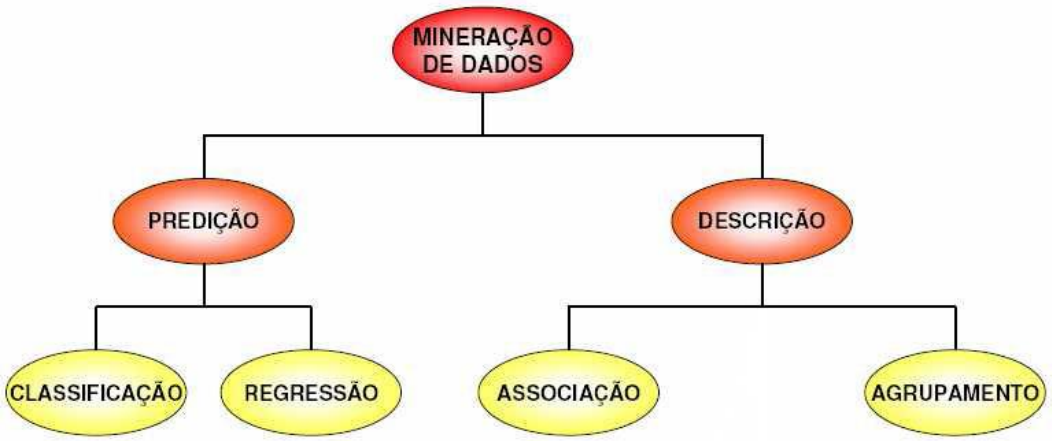


Figura 2.3: Principais tarefas de mineração de dados. [75]

As atividades de predição envolvem o uso dos atributos de um conjunto de dados para prever o valor futuro do atributo-meta, ou seja, essas atividades visam principalmente à tomada de decisões. Já as atividades de descrição procuram padrões interpretáveis pelos humanos que descrevem os dados antes de realizar a previsão. Essa tarefa também visa o suporte à decisão.

A predição consiste em examinar atributos de um conjunto de entidades e, baseado nos valores destes atributos, assinalar valores e atributos de uma nova entidade que se quer caracterizar. A predição usa atributos para predizer o desconhecido ou os valores futuros de outras variáveis. Os dois principais tipos de tarefas para predição são classificação e regressão.

De acordo com Berry e Linoff [7], a descrição tem por objetivo descrever o que está acontecendo em uma base de dados complicada no intuito de aumentar o entendimento sobre as pessoas, produtos ou processos que produziram os dados. Atividades de descrição consistem na identificação de comportamentos intrínsecos do conjunto de dados, sendo que estes dados não possuem uma classe especificada. Algumas das tarefas de descrição são agrupamento e regras de associação.

A seguir, são apresentados alguns conceitos de AM e também as técnicas de AM utilizadas nesse trabalho. Juntamente é apresentado um breve resumo sobre agrupamento, classificação e regressão, ou seja, as tarefas de mineração de dados das quais derivam as técnicas.

2.1.2 Aprendizado de Máquina

De acordo com Carbonell et al. [12] e Mitchell [62], o Aprendizado de Máquina (AM) é o campo da Inteligência Artificial que pode ser entendido como um conjunto de métodos, técnicas e ferramentas próprias para aquisição automatizada de conhecimento a partir de um conjunto de dados, melhorando seu desempenho por meio da experiência.

Witten e Frank [93], Carbonell et al. [12] e Mitchell [62] explicam que o sistema de AM pode ser classificado de duas maneiras: supervisionado e não-supervisionado. O sistema supervisionado diz que o algoritmo de aprendizado (indutor) recebe um conjunto de exemplos de treinamento para os quais os rótulos da classe associada são conhecidos. Cada exemplo (instância) é descrito por um vetor de valores (atributos) e pelo rótulo da classe associada. Um indutor pode ser visto como um algoritmo de AM capaz de criar uma classificação a partir de um conjunto de exemplos. Seu principal objetivo está em extrair conceitos expressos em alguma linguagem, por exemplo, regras de produção ou árvores de decisão, capazes de serem aplicadas a novos casos. Para rótulos de classe discretos, esse problema é chamado de classificação e para valores contínuos, de regressão.

No aprendizado não-supervisionado, o indutor analisa os exemplos fornecidos e tenta determinar se alguns deles podem ser agrupados de alguma maneira, formando agrupamentos ou *clusters*. Após a determinação dos agrupamentos, em geral, é realizada uma

análise para determinar o que cada agrupamento significa no contexto do problema analisado.

Na aplicação deste projeto foi trabalhado tanto o aprendizado de máquina supervisionado quanto o não-supervisionado. Inicialmente, utilizou-se o aprendizado não-supervisionado por meio de técnicas de agrupamento para identificar classes de equivalência sobre os dados não rotulados. Em seguida, utilizou-se o aprendizado supervisionado para gerar regras de inferência sobre o conjunto de exemplos de treinamento rotulados obtido com as técnicas de agrupamento. Uma forma simplificada da classificação dos sistemas de AM é apresentada na Figura 2.4.

Figura 2.4: Classificação dos sistemas de AM [13]

2.1.3 Paradigmas de Aprendizado de Máquina

Os paradigmas de Aprendizado de Máquina definem as diferentes formas de aprendizado automático. Os paradigmas são:

1. **Simbólico:** de acordo com Monard et al. [63], este paradigma busca aprender construindo representações simbólicas de um conceito através da análise de exemplos e contra-exemplos desse conceito. Atualmente, entre as representações simbólicas mais estudadas estão as árvores de decisão. As árvores de decisão utilizam um tipo de algoritmo de AM baseado na abordagem de dividir para conquistar. Árvores de decisão ou de classificação são técnicas de indução usadas para descobrir regras de classificação para um atributo a partir da subdivisão sistemática dos dados contidos no repositório que está sendo analisado. Em uma árvore de decisão, um problema

complexo é decomposto em sub-problemas mais simples, tornando o problema mais fácil de ser analisado. Entre as vantagens encontradas em árvores de decisões podem-se destacar pouco tempo de processamento utilizado, a facilidade de compreensão do modelo, bem como identificar os atributos chaves no processo, e expressar facilmente as regras como instruções lógicas aplicadas diretamente aos novos registros.

2. **Estatístico:** Rezende [74] explica que neste paradigma de aprendizado, o objetivo é construir um modelo estatístico que represente o mais próximo possível o conceito induzido. Vários destes modelos são paramétricos, na qual é assumida alguma forma para o modelo e então os parâmetros são ajustados de forma a representar a melhor aproximação sobre o conjunto de dados. As redes neurais são consideradas por alguns autores como exemplo de aprendizado que utilizada o paradigma estatístico. Isto porque as redes neurais precisam ser ajustadas com valores de pesos para as ligações entre os "neurônios"da rede. A técnica de redes neurais será detalhada em uma seção mais adiante deste capítulo. Outro exemplo de método estatístico de aprendizado é o Aprendizado Bayesiano, que utiliza um modelo probabilístico baseado no conhecimento prévio do problema combinado com exemplos de uma base de treinamento para determinar a probabilidade final de uma hipótese.

3. **Baseado em Exemplos:** este paradigma utiliza da comparação das características do exemplo a ser classificado com uma base de exemplos já classificados, e então assume que a classe do exemplo é a mesma classe daqueles exemplos já conhecidos e que possuem as mesmas características. Quando os exemplos precisam estar carregados na memória para que o classificador possa classificar novos elementos, o sistema é dito *lazy*. O contrário destes sistemas são os sistemas ditos *eagers*. Neste caso, os exemplos são descartados logo após serem utilizados para induzir o conhecimento. Os dois algoritmos mais importantes deste paradigma são o *Nearest Neighbours (KNN-Algorithm)* (vizinhos mais próximos) [62] e Raciocínio Baseado em Casos (RBC) [74].

4. **Conexionista**: De acordo com Rezende [74], este paradigma é baseado nas inter-

conexões entres os atributos representantivos de uma classe. Por esta razão, Redes Neurais é a técnica mais conhecida deste paradigma [20, 46].

5. **Genético**: este paradigma utiliza de uma analogia ao modelo de seleção natural de Charles Darwin, na qual os elementos de uma população evoluem de geração em geração, selecionando naturalmente aqueles mais adaptados ao ambiente e descartando os mais fracos. O paradigma genético utiliza a "evolução"de uma população de classificadores que competem para fazer a predição, sendo selecionado aquele que apresentar os melhores resultados. Pode-se citar Programação Genética e Algoritmos Genéticos [42, 49].

É importante citar a existência da Lógica Nebulosa (*Fuzzy*) [26], a qual consiste em um sistema nebuloso à base de regras, composto de um conjunto de regras de produção que definem ações de controle. Outro tipo de algoritmo comum compreende a família de algoritmos de otimização, como exemplos têm-se a família de Algoritmos Gulosos (*Greedy*) [19] e o Algoritmo *Hill Climbing* [50]. A seguir são apresentadas as técnicas de AM pertinentes a esse trabalho.

2.2 Técnicas de Aprendizado de Máquina

De acordo com Costa et al. [20], uma característica importante dos algoritmos de Aprendizado de Máquina é a capacidade de revelar padrões/conhecimentos corretos relativos a uma base, mesmo que esta contenha dados com imperfeições ou ausência de algumas informações. Esta capacidade é complementada por técnicas de pré-processamento e transformação dos dados que são aplicadas nas bases de dados a fim de aumentar a qualidade dos mesmos. Realizadas as tarefas de tratamento e preparação dos dados, a próxima etapa é a aplicação dos algoritmos de aprendizado. Dentro dos paradigmas de aprendizagem de máquina, existem as técnicas ou algoritmos que implementam a forma de aprendizado. Algumas técnicas mais comuns são: árvores de decisão, algoritmos de agrupamento, programação genética e redes neurais. Neste trabalho destacam-se as técnicas de agrupamento e árvores de decisão.

2.2.1 Agrupamento

Segundo Han e Kamber [35], a descoberta de agrupamento, ou clusterização, consiste em dividir uma população heterogênea em subgrupos homogêneos, com base na semelhança entre os registros do subgrupo. Esta técnica agrupa informações homogêneas de grupos heterogêneos entre os demais e aponta o ítem que melhor representa cada grupo, permitindo, desta forma, perceber as características de cada grupo. Esta técnica se caracteriza por trabalhar sobre dados onde os rótulos das classes não estão definidos. Diferentemente da técnica de classificação, em que os dados de treinamento estão devidamente classificados e os rótulos das classes são conhecidos.

Uma taxonomia com relação aos métodos de agrupamento proposta por Han e Kamber [35], tipifica os algoritmos de agrupamento da seguinte maneira:

- Particionamento: o agrupamento se dá por sucessivas subdivisões do conjunto original de elementos de dados em n subconjuntos disjuntos que passam a representar os grupos encontrados;

- Hierárquicos: os elementos de dados são ligados uns aos outros numa estrutura de árvore, onde os ramos podem ser formados por aglomeração (a partir de suas "folhas") ou por divisão (a partir da "raiz"), segundo critérios de similaridade pré-estabelecidos;

- Baseados em densidade: utiliza métodos voltados para descoberta de grupos com formas arbitrárias, levando-se em conta as diferentes densidades do espaço amostral, onde o ruído é representado por regiões de baixa densidade;

- Baseados em grades: utiliza uma estrutura de dados na forma de uma grade com multiresolução usada para quantificar um espaço amostral em um número finito de células, sobre as quais são conduzidas todas as tarefas de clusterização; e

- Baseados em modelos: Os métodos baseados em modelos usam um modelo para cada *cluster*. Eles tentam otimizar a curva entre os objetos dados e algum modelo matemático. Um algoritmo baseado em modelo pode descobrir *clusters* construindo

uma função de densidade que reflete a distribuição espacial dos pontos de dados. Ele também conduz a um modo de determinar automaticamente o número de *clusters* baseado na estatística padrão, identificando ruídos no relatório e assim produzindo métodos de agrupamento robustos. Os métodos de agrupamento baseados em modelos seguem uma das duas principais abordagens: estatística e por rede neural.

A Figura 2.5 apresenta os principais métodos para a realização de clusterização, de acordo com a taxonomia proposta por Jain et al. [43] para os métodos tradicionais, englobando as técnicas utilizadas para esse fim. Outras técnicas mais recentes, tais como clusterização nebulosa, Redes Neurais Artificiais, Redes *Info-Fuzzy*, entre outras, apesar de não terem sido incluídas no esquema da Figura 2.5, podem ser vistas em [43].

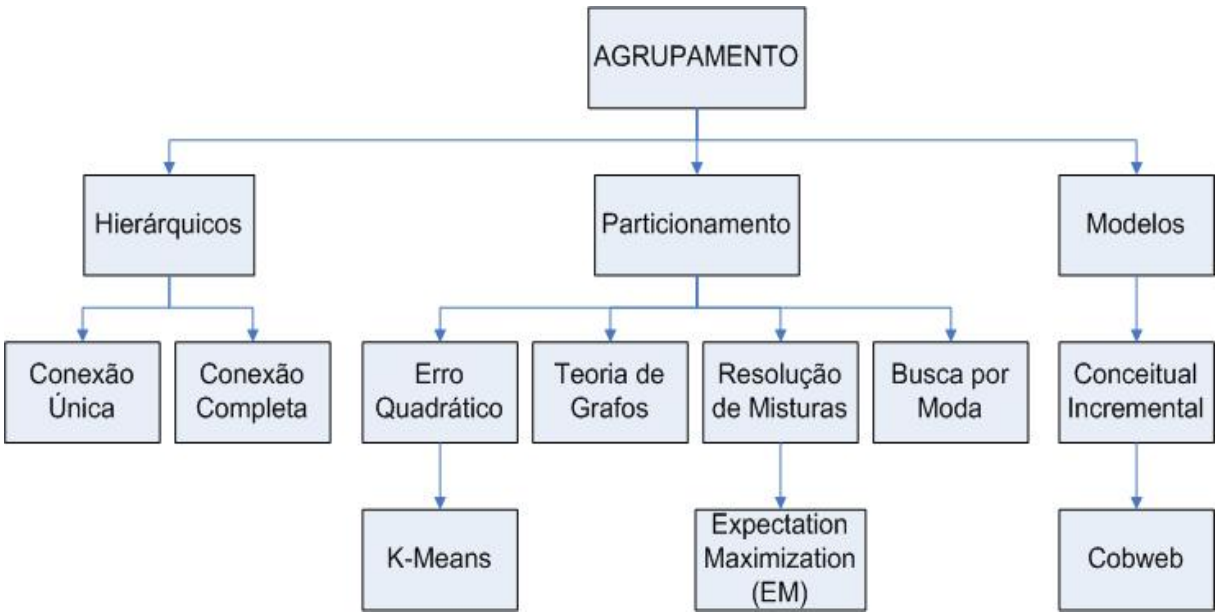


Figura 2.5: Taxonomia de abordagens de clusterização apresentada em [43]

Existem diversas taxonomias para os algoritmos de agrupamento e todas elas preservam coerência com relação aos mecanismos genéricos utilizados como critérios de tipificação. Assim, qualquer uma delas pode auxiliar na escolha daquele que melhor se ajusta ao domínio dos dados a serem agrupados, muito embora qualquer método de classificação não-supervisionada seja aplicável na resolução de qualquer problema de agrupamento.

De acordo com Han e Kamber [35], a aplicação dessas técnicas em grandes conjuntos de dados é capaz de revelar a formação de subgrupos de características que permitem o

desenvolvimento de novos esquemas de classificação sobre novas hipóteses de atributos. Este aspecto é bastante explorado no contexto da mineração de dados, já que a clusterização se realiza sobre elementos de dados representados por tuplas, onde a similaridade entre eles é, normalmente, definida a partir da verificação da proximidade entre os valores dos mesmos atributos que os descrevem, de acordo com uma determinada função de distância.

2.2.1.1 Uma Definição Formal Para o Problema de Clusterização

De acordo com Fränti e Kivijärvi [31], uma das formalizações para o problema de clusterização utiliza a seguinte terminologia para definir uma clusterização por particionamento:

- N: número de objetos de dados;
- K: número de *clusters*;
- M: número de atributos;
- X: conjunto de N objetos de dados $X = x_1, x_2, \cdots, x_N$;
- P: conjunto de N índices dos *clusters* $P = P_1, P_2, \cdots, P_N$;
- C: conjunto de K representantes dos *clusters* $C = C_1, C_2, \cdots, C_K$.

Dado um conjunto X com N objetos de dados (X_i), descritos por M atributos, particioná-lo em K *clusters*, de modo que os objetos mais similares estejam no mesmo grupo, enquanto os menos similares estejam em grupos diferentes. Um vetor (P) define a partição da clusterização ($P_i \cdots P_n$), fornecendo para cada objeto de dados o índice do seu respectivo *cluster* ($C_i \cdots C_K$), o qual é representado por um objeto de dados denominado centróide.

A escolha da função que formaliza uma solução para um problema de clusterização depende da aplicação, já que não existe uma medida que sirva como solução universal. Por exemplo, considerando, a soma do quadrado das distâncias dos objetos de dados para seus respectivos centróides (representantes dos *clusters*), então, dada uma partição P, o

conjunto C de centróides e a distância Euclidiana, uma partição pode ser expressa da seguinte forma:

$$f(P, C) = \sum_{i=1}^{N} d(x_i, Cp_i)^2 \tag{2.1}$$

2.2.1.2 *K-Means*

O *K-Means* é um algoritmo iterativo relativamente simples e muito empregado na clusterização não-supervisionada. Trata-se de uma heurística de busca local baseada em aprendizado competitivo que minimiza a função custo a partir de um conjunto inicial de centróides.

K-Means foi o primeiro e o mais famoso algoritmo de agrupamento, desenvolvido por Hartigan em 1975 [38]. Trata-se de um algoritmo no qual um conjunto de objetos é particionado em um conjunto fixo de grupos de maneira a formar grupos naturais, ou seja, de mínima variação intragrupo.

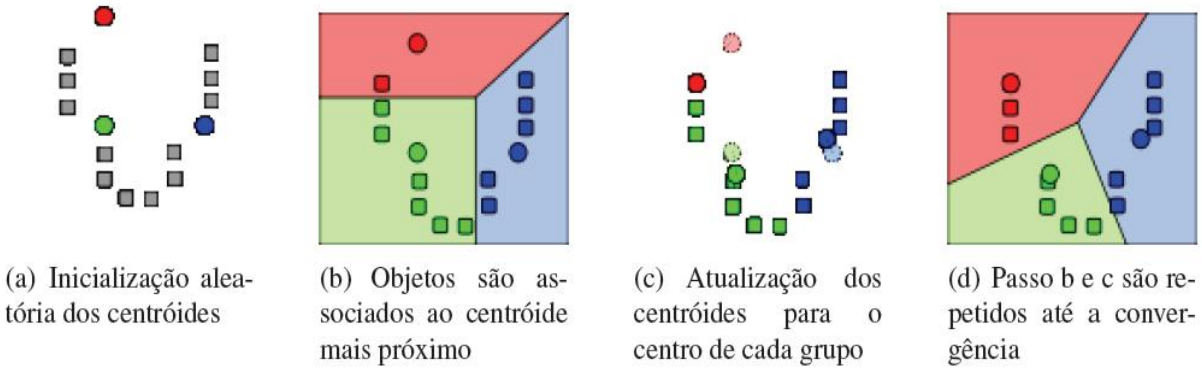
O critério de custo a ser minimizado é definido em função da distância dos elementos em relação aos centros dos agrupamentos. Usualmente, este critério é a soma residual dos quadrados das distâncias (geralmente é usada a distância euclidiana). Entende-se por soma residual dos quadrados, a soma dos quadrados das distâncias dos elementos ao centróide do seu *cluster*, conforme a equação:

$$W = \sum_{i=1}^{k} \sum_{j=1}^{n_i} (x_{ij} - X_i)(x_{ij} - X_i)^t \tag{2.2}$$

Onde x_{ij} é o j-ésimo objeto do *cluster* i , X_i é o representante do *cluster* i (a média ou mediana dos objetos do *cluster*), e n_i é a quantidade de objetos do *cluster* i . O elemento representativo de um *cluster* é o seu centróide, que possui um valor médio para os atributos considerados, relativos a todos os elementos do *cluster*. A utilização do centróide como

elemento representativo de um *cluster* é conveniente apenas para atributos numéricos e possui um significado geométrico e estatístico claro, podendo, entretanto, receber mais influência de um único elemento que se encontre próximo à fronteira do *cluster*.

A forma de solução proposta para este problema é baseada no algoritmo de Lloyd [57] e está ilustrada na Figura 2.6. O conjunto de objetos de entrada é particionado em conjuntos iniciais; esta partição pode se valer de alguma heurística ou simplesmente serem escolhidos aleatoriamente. É calculado o centróide de cada partição (a). Uma nova partição é construída associando-se cada objeto ao centróide mais próximo (b). Os centróides são atualizados para o centro de cada grupo (c), e o algoritmo repete a aplicação alternada dos procedimentos (b) e (c) até atingir a convergência (d), a qual é obtida quando os centróides não são mais alterados.



(a) Inicialização aleatória dos centróides

(b) Objetos são associados ao centróide mais próximo

(c) Atualização dos centróides para o centro de cada grupo

(d) Passo b e c são repetidos até a convergência

Figura 2.6: Demonstração do algoritmo K-Means

O agrupador *K-Means* apresenta como principais características o favorecimento de grupos de formato esférico e o particionamento rígido dos grupos, ou seja, cada objeto pertence, integralmente, a um e somente um grupo. As vantagens desse agrupador são a simplicidade matemática, a rápida convergência do algoritmo de minimização e escalabilidade em relação aos dados de entrada. Apesar da simplicidade, o algoritmo apresenta muitas desvantagens, pois não possui imunidade a ruídos, é necessário especificar o número de grupos, favorece a formação de grupos esféricos e, principalmente, o algoritmo de minimização pode estar sujeito a mínimos locais.

2.2.1.3 *EM* (*Expectation-Maximization*)

De acordo com Witten e Frank [93], o algoritmo de geração de estimativas de máxima verossimilhança é o *EM* (*Expectation-Maximization*), ele depende de parâmetros iniciais para que o processo de iteração seja iniciado. Este processo de iteração é necessário para que possam ser aproximadas soluções de equações não lineares, a partir de um valor inicial dado, até que haja convergência a um valor final. O objetivo é haver convergência até um valor máximo global. Contudo, dependendo dos parâmetros iniciais, pode-se obter um máximo local iterativamente, o que não é o desejado.

Dado que o algoritmo *EM* procura encontrar o melhor ajuste de um modelo para um certo conjunto de dados, deve haver alguma forma de se avaliar a adequação deste modelo ajustado aos dados. A premissa básica do algoritmo, assim, é que deve existir um ponto ótimo a partir do qual não é preciso mais qualquer tentativa de um novo modelo, com novos parâmetros e, neste momento, as iterações terminam. Dessa forma, o algoritmo *EM* continua o processo de convergência rumo a uma solução até que a mudança de probabilidade entre dois conjuntos de estimativas seja desprezível, a partir de critérios de convergência previamente definidos. É comum que os usuários destes programas computacionalmente intensivos não tenham claros os problemas decorrentes de se encontrar um máximo local em vez de um máximo global.

A impossibilidade de se garantir a ocorrência de um fato, mesmo que seja bastante provável, requer a realização de ajustes às equações de estimativa de parâmetros, de modo que, se w_i é a probabilidade de pertinência da instância i ao *cluster* A, a média e o desvio padrão para o *cluster* A são:

$$\mu_A = \frac{w_1 x_1 + w_2 x_2 + \cdots + w_n x_n}{w_1 + w_2 + \cdots + w_n} \tag{2.3}$$

$$\sigma_A^2 = \frac{w_1 (x_1 - \mu)^2 + w_2 (x_2 - \mu)^2 + \cdots + w_n (x_n - \mu)^2}{w_1 + w_2 + \cdots + w_n} \tag{2.4}$$

onde a variável x_i representa todos os elementos e não somente aqueles que realmente pertencem ao *cluster* A, o que pode ser visto como um estimador de probabilidade máxima para a variância.

A qualidade da clusterização aumenta a cada iteração do algoritmo geral e pode ser medida pela multiplicação das probabilidades individuais de pertinência de cada elemento i a cada *cluster* K, conforme a expressão abaixo, onde as probabilidades dos *clusters* $P_{C_{1...k}}$ são determinadas pela função de distribuição normal $f(x; \mu, s)$.

$$\Pi_i = (P_{C_1} P[x_i|C_1] + ... + P_{C_k} P[x_i|C_k]) \tag{2.5}$$

Enquanto *K-Means* alcança um ponto fixo, no qual as classes dos elementos não mais se modificam entre uma iteração e outra, no *EM* isso pode não acontecer e, portanto, algum critério de parada deve ser estabelecido. Como exemplo de um critério de parada, pode ser estabelecido que isto deva acontecer quando a diferença entre os valores computados pela expressão citada acima for inferior à 10^{-10}, após 10 iterações sucessivas.

Em linhas gerais o algoritmo EM realiza o seguinte procedimento:

1. Estabelece-se um palpite inicial para os parâmetros das fórmulas da média e desvio padrão;

2. Enquanto houver melhoria de qualidade:

3. computar a probabilidade dos *clusters* para cada elemento de dados;

4. estimar os novos valores dos seis parâmetros, utilizando os resultados das probabilidades obtidas no passo anterior.

A melhoria citada no passo 2 é verificada pelo aumento do valor computado para a expressão 2.5, enquanto a probabilidade do passo 3 pode ser obtida pela computação da função de densidade de probabilidade, apresentada em 2.6, para o caso de uma única variável independente x com média μ e desvio padrão *sigma*.

$$f(x) = \frac{1}{(\sqrt{2\pi\sigma})e^{\frac{-(x-\mu)^2}{2\sigma^2}}} \tag{2.6}$$

Apesar da convergência garantida do EM, não se pode dizer que o ponto de parada é um máximo global e, por isso, a utilização de outras heurísticas/metaheurísticas pode aumentar a chance de melhores resultados. Por exemplo, realizar várias repetições do procedimento inteiro a partir de palpites iniciais aleatórios não deixa de ser uma heurística simples cujo objetivo seria a maximização do valor que representa a qualidade da clusterização. A aplicação deste conceito, com relação ao algoritmo *EM* implementado no *Weka*, é obtida através da validação cruzada, a qual é utilizada para determinar o número de *cluster* da seguinte maneira:

1. Inicialmente, o número de *clusters* K passa a ser igual a 1;
2. Os elementos da amostra são distribuídos randomicamente entre as décimas partes da amostra;
3. Executa-se o algoritmo EM 10 vezes, variando-se as partes selecionadas para treinamento de acordo com a maneira usual da *10-fold cross validation*;
4. Uma média das probabilidades é estabelecida para todos os resultados obtidos;
5. Caso o valor das médias das probabilidades tenha aumentado com relação ao obtido na iteração anterior, então $K = K + 1$ e retorna-se ao passo 2.

2.2.1.4 *COBWEB*

De acordo com Han e Kamber [35], o agrupamento conceitual é uma forma de agrupamento em AM que, dado um conjunto de objetos não rotulados, produz um esquema de classificação sobre os objetos. Ao contrário das técnicas de agrupamento convencionais, que primariamente identificam grupos de objetos, o agrupamento conceitual realiza uma etapa adicional para encontrar descrições características para cada grupo, onde cada

grupo representa um conceito ou classe. Por isso, o agrupamento conceitual é um processo de duas etapas: primeiro, o agrupamento é realizado, seguido pela caracterização. Aqui, a qualidade do agrupamento não é unicamente uma função dos objetos individuais. Antes, ele incorpora fatores tais como a generalidade e a simplicidade das descrições conceituais derivadas.

Muitos algoritmos de agrupamento adotam uma abordagem estatística que usa medidas de probabilidade na determinação dos *clusters* ou conceitos. Descrições probabilísticas são, tipicamente, usadas para representar cada conceito derivado. O *COBWEB* é um algoritmo popular e simples de agrupamento conceitual incremental. Seus objetos de entrada são descritos por pares de valores de atributos categóricos. O algoritmo *COBWEB* cria um agrupamento hierárquico na forma de uma árvore de classificação.

Uma árvore de classificação difere de uma árvore de decisão. Cada nó em uma árvore de classificação refere-se a um conceito e contém uma descrição probabilística daqueles conceitos que resumem os objetos classificados abaixo do nó. A descrição probabilística inclui a probabilidade do conceito e probabilidades condicionais da forma $P(A_i = V_{ij}|C_k)$, onde $A_i = V_{ij}$ é um par de valores de atributos e C_k é a classe do conceito. Contadores são acumulados e armazenados em cada nó para computação das probabilidades. Isto é diferente de árvores de decisão, que rotulam ramos melhor do que nós e usam lógica melhor do que descritores probabilísticos. Para classificar um objeto usando uma árvore de classificação, uma função parcial é empregada para descer a árvore ao longo de um caminho de melhores nós.

O algoritmo *COBWEB* usa uma medida de avaliação estatística chamada utilidade categórica para guiar a construção da árvore. A utilidade categórica é definida como:

$$UA = \frac{\sum_{k=1}^{n} P(C_k)[\sum_i \sum_j P(A_i = V_{ij}|C_k)^2 - \sum_i \sum_j P(A_i = V_{ij})^2]}{n} \tag{2.7}$$

onde n é o número de nós, conceitos ou categorias formando uma partição $C_1, ..., C_n$ de um dado nível da árvore. Em outras palavras, utilidade categórica é o aumento no número

esperado de valores de atributos que podem ser corretamente previstos para uma dada partição (onde este número esperado corresponde ao termo $P(C_k)$ vezes a primeira parcela entre colchetes) sobre o número esperado de previsões corretas sem tal conhecimento (correspondendo ao segundo termo entre colchetes).

A utilidade categórica recompensa a similaridade intraclasse e a dissimilaridade interclasse onde:

- a similaridade intraclasse é a probabilidade $P(A_i = V_{ij}|C_k)$.
- a dissimilaridade interclasse é a probabilidade $P(C_k|A_i = V_{ij})$.

O *COBWEB* incorpora, incrementalmente, objetos em uma árvore de classificação. Desce a árvore ao longo de um caminho apropriado, atualizando as quantidades ao longo do caminho, na busca do melhor hospedeiro ou nó para classificar o objeto. A decisão do caminho apropriado é baseada na localidade temporária do objeto em cada nó e pelo cálculo da utilidade categórica da partição resultante. A colocação que resulta na mais alta categoria deve indicar um bom hospedeiro para o objeto.

O algoritmo *COBWEB* também computa a utilidade categórica da partição que resultaria se um novo nó fosse ser criado para o objeto. Isto é comparado à computação anterior baseada nos nós existentes. O objeto é, então, colocado em uma classe existente, ou uma nova classe é criada para ele, baseado na partição com o mais alto valor de utilidade categórico. Nota-se que o algoritmo *COBWEB* tem a habilidade de automaticamente ajustar o número de classes em uma partição. Ele não precisa contar com o usuário para prover tal parâmetro de entrada. Os dois operadores mencionados acima são altamente sensíveis à ordem de entrada dos objetos.

O algoritmo tem dois parâmetros adicionais que ajudam a fazê-lo menos sensível à ordem de entrada. Estes são os parâmetros juntar e separar. Quando um objeto é incorporado, os dois melhores hospedeiros são considerados para juntar em uma simples classe. Além disso, o *COBWEB* considera separar os filhos do melhor hospedeiro entre as categorias existentes. Estas decisões são baseadas na utilidade categórica. Os operadores

de juntar e separar permitem ao algoritmo desempenhar uma busca bidirecional, onde um ajuntamento pode desfazer uma divisão prévia.

O algoritmo *COBWEB* possui várias limitações. Primeiro, ele é baseado na suposição que as distribuições de probabilidade dos atributos são estatisticamente independentes entre si. Esta suposição é, entretanto, nem sempre verdadeira visto que a correlação entre atributos freqüentemente existe. Além disso, a representação da distribuição de probabilidade de *clusters* torna o algoritmo muito custoso para atualizar e armazenar os *clusters*. Isto ocorre especialmente quando os dados têm um grande número de atributos, visto que suas complexidades de tempo e espaço dependem não somente do número de atributos, mas também do número de valores para cada atributo. Além disso, a árvore de classificação não é balanceada na altura para os dados de entrada, o que pode degradar a complexidade de tempo e espaço dramaticamente.

2.2.1.5 Comparativo Entre os Algoritmos Apresentados

A seguir são listadas as principais vantagens e desvantagens, assim como uma breve análise dos parâmetros dos algoritmos apresentados anteriormente:

- K-Means - Vantagens
  - Simplicidade matemática;
  - Rápida convergência do algoritmo de minimização;
  - Escalabilidade em relação aos dados de entrada.
- K-Means - Desvantagens
  - Não possui imunidade a ruídos;
  - É necessário especificar o número de grupos;
  - Favorece a formação de grupos esféricos;
  - É sujeito a mínimos locais.
- K-Means - Parâmetros

  - Clusters: Somente um parâmetro requerido, que fixa o número de clusters.

- EM - Vantagens
  - Utiliza validação cruzada para determinar o número de clusters.

- EM - Desvantagens
  - Pode-se obter um máximo local, dependendo dos parâmetros iniciais.

- EM - Parâmetros
  - MinStdDev: Critério de parada;
  - MaxIterations: Número de iterações que o algoritmo é executado.

- COBWEB - Vantagens
  - Utiliza agrupamento conceitual;
  - Utiliza árvore de classificação e cálculo de utilidade categórica.

- COBWEB - Desvantagens
  - É baseado na suposição que as distribuições de probabilidade dos atributos são estatisticamente independentes entre si;
  - A representação da distribuição de probabilidade de clusters torna o algoritmo muito custoso para atualizar e armazenar os clusters;
  - A árvore de classificação não é balanceada na altura para os dados de entrada, o que pode degradar a complexidade de tempo e espaço dramaticamente.

- COBWEB - Parâmetros
  - Cluster: Possibilidade de fixar ou não o número de clusters;
  - Cutoff: Determina o nível de semelhança de um atributo;
  - Acuity: Esse parâmetro corresponde à noção da menor diferença perceptível nos valores e é utilizado quando o desvio padrão é zero para um conceito.

2.2.2 Classificação

Weiss e Indurkhya [92] explicam que classificação é o processo de encontrar um conjunto de modelos que descrevem e distinguem classes, com o propósito de utilizar o modelo final (refinado) para predizer a classe de objetos que ainda não foram classificados. O modelo construído baseia-se na análise prévia de um conjunto de dados de amostragem ou dados de treinamento, contendo objetos corretamente classificados. A classificação consiste na predição de um valor categórico como, por exemplo, predizer a cobertura ou não de uma classe de defeitos. Na regressão, o atributo a ser predito consiste em um valor contínuo como, por exemplo, predizer a porcentagem de cobertura para um determinado critério de teste. Árvore de Decisão é uma técnica de classificação largamente utilizada.

2.2.2.1 Árvore de Decisão

De acordo com Apte e Weiss [3], Árvores de Decisão ou de classificação são técnicas usadas para descobrir regras de classificação para um atributo a partir da subdivisão sistemática dos dados contidos no repositório que está sendo analisado. São simples representações de conhecimento e classificam exemplos em um número finito de classes.

A árvore de decisão consiste de uma hierarquia de nós internos e externos que são conectados por ramos (arcos). O nó interno é a unidade de tomada de decisão que avalia por meio de teste lógico qual será o próximo nó descendente ou filho. Em contraste, um nó externo (não tem nó descendente), também conhecido como folha, está associado a um rótulo ou a um valor, que indica a classe predita para um determinado conjunto de dados.

Breiman et al. [10] diz que, em geral, o procedimento de uma árvore de decisão é o seguinte: apresenta-se um conjunto de dados ao nó inicial (ou nó raiz que também é um nó interno) da árvore; dependendo do resultado do teste lógico usado pelo nó, a árvore ramifica-se para um dos nós filhos e este procedimento é repetido até que um nó folha é alcançado. A repetição deste procedimento caracteriza a recursividade da árvore de decisão.

Os classificadores geralmente são construídos por programas chamados indutores, que implementam algoritmos computacionais especiais, que operam sobre uma massa de dados

inicial considerada representativa do domínio do problema e na qual tanto o valor dos atributos comuns quanto da classe de cada objeto são conhecidos. Um programa indutor de classificadores procurará, com base nas ocorrências desse conjunto de dados inicial, chamado de conjunto de treinamento, estabelecer qual a ligação entre os valores dos atributos não-categóricos e as classes encontradas na massa de dados.

Conforme Weiss e Indurkhya [92], a expectativa é de que essas relações encontradas, chamadas de regras de classificação e que representam em última instância o classificador em si, possam ser empregadas posteriormente para determinar o valor da classe para objetos onde essa informação é desconhecida, num tipo de atividade chamada predição (da classe).

Para que o grau de acerto de um classificador assim produzido possa ser avaliado antes de sua efetiva utilização prática, geralmente procura-se aplicá-lo sobre um segundo conjunto de dados onde o valor da classe é igualmente conhecido, chamado conjunto de teste. O conjunto de teste possui valores para os atributos de predição e também valores de classe para cada caso, ou seja, é estruturalmente idêntico ao conjunto de treinamento, mas seus dados não são os mesmos.

Posteriormente, compara-se o grau de concordância entre a classe prevista pelo classificador para cada objeto e a classe realmente observada. Sendo assim, dados um conjunto de exemplos de tamanho finito, um conjunto de treinamento e um indutor, é importante estimar o desempenho futuro do classificador induzido utilizando o conjunto de exemplos. A partir do conjunto de treinamento, treina-se um indutor e testa-se seu desempenho com esse conjunto teste, que são exemplos fora da amostra utilizada para treinamento.

A respeito dos algoritmos, muitos são os algoritmos de classificação que elaboram árvores de decisão. Não há uma forma de determinar qual é o melhor algoritmo, um pode ter melhor desempenho em determinada situação e outro algoritmo pode ser mais eficiente em outros tipos de situações.

De acordo com Carbonell et al. [12], a forma mais tradicional para a indução de regras de classificação baseia-se em uma estratégia de divisão-e-conquista conhecida por TDIDT (*Top-Down Induction of Decision Trees*) ou *ID3*. Entre os principais algoritmos

de indução de regras de classificação, destacam-se o *ID3, C4.5, C5.0* e *J48*, comentados a seguir:

O algoritmo *ID3* [71] foi um dos primeiros algoritmos de árvore de decisão, tendo sua elaboração baseada em sistemas de inferência e em conceitos de sistemas de aprendizagem. Logo após foram elaborados diversos algoritmos, sendo os mais conhecidos: *C4.5* [71], *CART* (*Classification and Regression Trees*), *CHAID* (*Chi Square Automatic Interaction Detection*), entre outros.

Conforme Quinlan [71], o algoritmo *ID3* inicial foi posteriormente aperfeiçoado, dando origem ao *C4* e *C4.5*, que estenderam suas funcionalidades ao permitir o tratamento de conjuntos de treinamento com valores de atributo desconhecidos; valores de atributos contínuos; implementação de estratégias de poda de árvore e recursos para a extração de regras SE-ENTÃO a partir da árvore inicialmente induzida. É provavelmente a família de algoritmos de indução de regras mais conhecida na área do aprendizado computacional. Mais recentemente uma nova geração desse algoritmo foi desenvolvida, oferecendo diversos aprimoramentos e passando a ser oferecida como um produto comercial denominado *C5.0*.

Ainda para o mesmo autor, o *C4.5* é uma melhora do *ID3*, ou seja, além de possuir as mesmas características, ele possui a vantagem de poder lidar com a poda da árvore para evitar o *overfitting*, com a ausência de valores e com a presença de ruídos nos dados. O algoritmo J48 é uma reimplementação em Java do algoritmo C4.5 [71] e faz parte do pacote de algoritmos de AM da ferramenta *Weka* [91].

O *J48* constrói um modelo de árvore de decisão baseado num conjunto de dados de treinamento, e usa esse modelo para classificar outras instâncias num conjunto de teste. Durante o processo de utilização do algoritmo *J48* é interessante conhecer alguns parâmetros que podem ser modificados para proporcionar melhores resultados como, por exemplo, o uso de podas na árvore, o número mínimo de instâncias por folha e a construção de árvore binária.

A escolha central de um algoritmo está em selecionar qual atributo será usado em cada nó da árvore. É interessante selecionar o atributo que é mais útil para classificar exemplos. Em outras palavras, uma boa subdivisão no momento da construção da árvore, é aquela

que produz para os dados disponíveis os grupos mais homogêneos com relação ao atributo classe, enquanto que as más subdivisões caracterizam-se por formar grupos com pouca identidade com relação à classe. Assim, é definida uma propriedade estatística chamada ganho de informação, que mede como um determinado atributo separa os exemplos de treinamento de acordo com a classificação deles. É utilizado o ganho de informação para selecionar, entre os candidatos, os atributos que serão utilizados a cada passo, enquanto constrói a árvore. O melhor atributo é aquele com o ganho de informação maior.

Resumidamente, a estratégia para determinar o melhor atributo de divisão é expressa em calcular o ganho de informação para cada atributo não categórico disponível e escolher aquele que apresentar o maior valor, descartando-o em seguida do processo de escolha para os próximos níveis da árvore de decisão. O interessante é descobrir o atributo que forme os subconjuntos mais homogêneos no que diz respeito ao atributo classe. Em outras palavras, aquele que forma os grupos menos confusos com relação à classe.

2.3 A Ferramenta *Weka*

A ferramenta *Weka* (*Waikato Enviroment for Knowledge Analysis*) [91] foi desenvolvida na Universidade de Waikato, Nova Zelândia. Este software implementa um grande conjunto de ferramentas para preparação dos dados, AM e validação/verificação dos resultados.

Inicialmente, as aplicações do *Weka* foram direcionadas para problemas de agricultura, no entanto, sua aplicação estendeu-se para diversas áreas onde há a necessidade da mineração de dados e descoberta de padrões em grandes quantidades de dados. Por ter sido desenvolvido por uma universidade, a maioria dos módulos do sistema são resultados de teses e dissertações na área de descoberta de conhecimento. Outro fator importante para sua aceitação e utilização em diversos trabalhos acadêmicos é a linguagem Java que foi utilizada para sua construção, permitindo ao sistema um bom nível de portabilidade.

O *Weka* dispõe de uma interface gráfica para a maioria de suas funções, permitindo ao usuário a seleção da fonte de dados, remoção dos atributos e seleção automática e manual de atributos que serão utilizados na descoberta de conhecimento. Além da preparação

dos dados, o *Weka* oferece alguns algoritmos classificadores como o *ZeroR*, o *Id3*, o *PART* e o *J48*, algoritmos de descoberta de regras de associação como o *Apriori* e algoritmos de agrupamento como o *K-Means*, *COBWEB* e *EM*. Além da interface gráfica, o *Weka* disponibiliza uma série da *APIs* (*Application Program Interface*) que permitem a utilização de seus algoritmos.

2.4 Considerações Finais

Este capítulo apresentou os principais conceitos e técnicas relacionados ao Aprendizado de Máquina. Também foram expostas, com maior nível de detalhe, as técnicas de agrupamento e classificação utilizadas neste estudo. Os próximos capítulos tratam da atividade de teste de software e de técnicas de teste de regressão, além de apresentar os trabalhos relacionados.

CAPÍTULO 3

FUNDAMENTOS DO TESTE DE SOFTWARE

Neste capítulo, são abordados alguns conceitos sobre a atividade de teste de software, objetivos, principais fases, técnicas e critérios que podem ser utilizados.

3.1 Terminologia e Conceitos Básicos

Existem várias tentativas no sentido de definir a atividade de teste, desde a visão intuitiva até a visão formal [5, 41, 64]. Segundo Myers [64], o teste de software consiste em executar o programa fornecendo os dados de teste necessários para a execução. A partir daí, obtém-se a saída gerada, a qual deve ser comparada com a saída esperada. Esta, por sua vez, é obtida da especificação do software. A comparação da saída gerada com a saída esperada é realizada com a intenção de revelar defeitos. O objetivo principal do teste é identificar o número máximo de defeitos dispondo do mínimo de esforço. O par formado pelo dado de teste e sua saída esperada forma um caso de teste.

Para tanto torna-se necessária a definição dos conceitos de defeito, engano, erro e falha. Conforme o padrão IEEE número 610.12-1990 [1], têm-se que:

- defeito (*fault*): passo, processo ou definição de dados incorreto, como por exemplo, uma instrução ou comando incorreto;
- engano (*mistake*): ação humana que produz um resultado incorreto, como por exemplo, uma ação incorreta tomada pelo programador;
- erro (*error*): diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução do programa constitui um erro; e
- falha (*failure*): produção de uma saída incorreta com relação à especificação.

Na atividade de teste, pressupõe-se a existência de um oráculo, o testador ou algum outro mecanismo. Este tem a função de determinar se o programa comporta-se conforme o esperado para todos os dados de entrada, dentro de limites de tempo e esforço [24]. Contudo, o teste não pode mostrar a ausência de defeitos, somente pode indicar se eles estão presentes [70]. Apesar deste fato, os testes contribuem para aumentar a confiança de que o software desempenha corretamente os requisitos especificados, desde que sejam conduzidos sistemática e criteriosamente [24].

A execução exaustiva do software seria a melhor maneira para revelar todos os defeitos. Contudo, essa execução toma todos os dados do domínio de entrada para serem testados, tornando inviável a execução desse tipo de teste, devido às restrições de tempo e custo. Além disso, ainda existem programas onde o domínio de entrada é infinito, impossibilitando a execução exaustiva. Dadas as circunstâncias, a seleção de casos de teste assume um papel crucial para um teste bem sucedido [60].

Outro fator relevante é referente à suficiência e qualidade dos testes realizados. Para tanto são utilizados critérios de teste com o intuito de auxiliar a elaboração de bons casos de teste e decidir se o software foi suficientemente testado. Sendo assim, quando a execução de casos de teste satisfizer um determinado critério selecionado, dá-se por encerrada a fase de testes [24]. Os critérios de teste definem elementos no programa, constituindo requisitos que devem ser satisfeitos, os quais são chamados elementos requeridos.

O objetivo é avaliar a própria aplicação dos testes garantindo que um conjunto específico de características foi exercitado. A função dos critérios de teste é estabelecer o que deve ser testado no software. Ou seja, fornecer uma maneira rigorosa e sistemática para selecionar casos de teste de forma a aumentar as chances de revelar defeitos e estabelecer medidas de confiabilidade com relação à corretude do programa. E no caso de defeitos não serem revelados, estabelecer um nível de confiabilidade elevado em relação à corretude do programa.

Assim, pode-se identificar o conjunto de elementos requeridos pelo critério utilizado no teste. Conhecendo os elementos requeridos a serem testados, um subconjunto do domínio

de entrada pode ser selecionado para satisfazer tais elementos, objetivando o aumento de produtividade do teste. A geração de dados de teste para satisfazer um determinado critério pode ser realizada utilizando informações de código-fonte, especificação do software, informações históricas de defeitos comuns, ou ainda, gerados aleatoriamente [60].

As propriedades mínimas que devem ser preenchidas por um critério de teste são [24]:

- garantir, do ponto de vista de fluxo de controle, a cobertura de todos os desvios condicionais;
- requerer, do ponto de vista de fluxo de dados, ao menos um uso de todo resultado computacional;
- requerer um conjunto finito de casos de teste.

3.2 Estratégia para Teste

A estratégia de teste em formato espiral proposta por Pressman [70] é composta de quatro fases, conforme exposto na Figura 3.1.

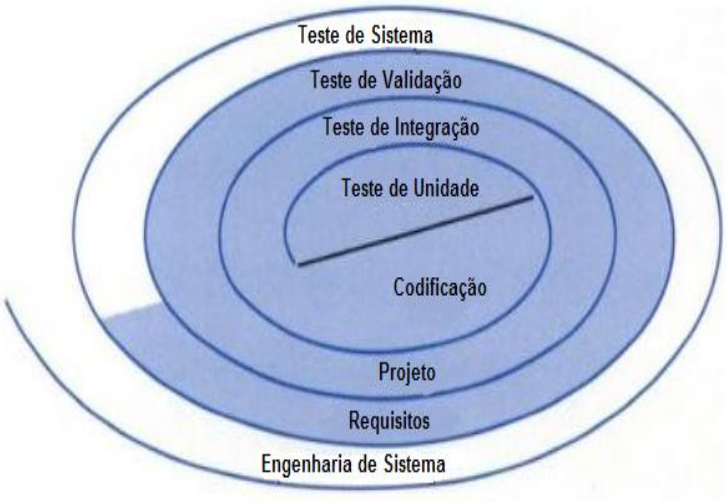


Figura 3.1: Estratégia de teste de software adaptada de [70]

- Teste de Unidade: É a aplicação de teste nas menores unidades de software que foram implementadas, o módulo. Visa o exame da estrutura de dados local e a identificação de erros de lógica e implementação;

- Teste de Integração: Na fase de teste de integração o objetivo é testar a integração dos vários módulos e verificar se eles funcionam juntos. Esta fase visa à identificação de erros de interface entre os módulos. Existem duas abordagens para este tipo de teste: incremental e não-incremental, sendo que a abordagem incremental é realizada por meio da integração módulo a módulo e a não incremental testa a integração de todos os módulos do sistema de uma só vez. A abordagem incremental ainda pode ser subdividida em *bottom-up* e *top-down*. Na abordagem *bottom-up* a integração ocorre, primeiramente, nos níveis hierárquicos inferiores e, então, vai subindo na hierarquia até que o último módulo seja integrado. Para tanto, os *drivers* vão sendo substituídos e combinados. Já na abordagem *top-down* a integração começa no nível mais alto e vai descendo até alcançar os módulos inferiores. Sendo assim, os *stubs* vão sendo substituídos por programas reais. *Drivers* e *stubs* simulam módulos do sistema que ainda não foram desenvolvidos para possibilitar a troca de dados entre os módulos;

- Teste de Validação: Esta fase é responsável por mostrar que o software opera corretamente e de acordo com os critérios de validação do usuário. Duas abordagens geralmente são utilizadas para o teste de validação: *alfa* que é realizado pelo usuário num ambiente controlado pelo desenvolvedor e *beta* que é realizado pelo usuário em seu ambiente, não controlado pelo desenvolvedor;

- Teste de Sistema: Visa o teste do software com todos os elementos, verificando se foram adequadamente integrados e cumprem os requisitos funcionais. Testa funcionalidade, desempenho, carga, segurança e situações anormais do sistema.

A estratégia de teste sempre é iniciada no centro do espiral com o teste de unidade que é concentrado em testar cada unidade do código-fonte, o módulo. Movendo-se ao longo do espiral, executa-se o teste de integração que está alinhado ao projeto do software, visando identificar problemas na interface entre os módulos. Em seguida, tem-se a validação que é responsável por confrontar o levantamento de requisitos, verificando se os requisitos solicitados pelo usuário estão corretos no software. Por fim, tem-se o teste de sistema

que é responsável por testar o sistema como um todo, verificando segurança, desempenho entre outros requisitos [70].

Qualquer estratégia de teste deve incorporar as atividades de planejamento, projeto de casos de teste, execução e avaliação dos resultados dos testes [5, 24, 60, 64, 70]. Essas atividades devem ser desenvolvidas ao longo do próprio processo de desenvolvimento do software.

3.3 Técnicas e Critérios de Teste

Os critérios de teste de software derivam, basicamente, de três técnicas de teste: a funcional, a estrutural e a baseada em defeitos. O que diferencia uma técnica da outra é a origem da informação utilizada na construção dos conjuntos de casos de teste, sendo que cada técnica possui uma variedade de critérios [60].

3.3.1 Técnica Funcional

Na técnica funcional utiliza-se a especificação ou requisitos do software para derivar os dados de teste sem levar em consideração a estrutura do código-fonte do programa. Seu objetivo é determinar se o programa satisfaz os requisitos funcionais especificados. Essa técnica indica que deve-se derivar conjuntos de condições de entrada, que devem exercitar plenamente todos os requisitos funcionais do programa. Segundo Myers [64], essa técnica é conhecida como caixa-preta porque só é possível visualizar o lado externo, ou seja, é fornecido o conjunto de dados de entrada ao programa e, então, verifica-se as respostas produzidas como saída, sem entretanto, verificar detalhes da implementação.

Uma das principais dificuldades relativas à técnica funcional é referente à especificação que, muitas vezes, é feita de modo descritivo e não formal. Esse fato faz com que os requisitos de teste também sejam imprecisos e informais, tornando difícil a automação da aplicação de critérios funcionais. Contudo, somente são necessárias a identificação das entradas, a função a ser computada e a saída do programa, facilitando a aplicação dessa técnica em todas as fases do teste (unidade, integração, validação e sistema) [22].

Pressman [70] diz que a técnica funcional consiste em duas etapas: identificar as funções que o software deve realizar e criar casos de teste capazes de checar se essas funções estão sendo realizadas corretamente. Visto que as funções do software são obtidas da especificação, é de suma importância uma especificação bem elaborada e consistente com os requisitos do usuário para um teste adequado.

As seguintes categorias de defeitos são o foco do teste funcional [70]: funções incorretas ou omitidas; erros de interface; erros de estrutura de dados ou de acesso à base de dados externa; erros de comportamento ou de desempenho; e erros de iniciação e término.

Como exemplos de critérios de teste funcional têm-se [70]:

3.3.1.1 Particionamento em Classes de Equivalência

Esse critério de teste propõe a divisão do domínio de entrada de um programa em classes de equivalência, a partir das quais os casos de teste são derivados. As entradas do programa são agrupadas em classes de acordo com a funcionalidade e tipo de defeitos. O critério supõe que as entradas (dados de teste) na mesma classe têm a mesma probabilidade de revelar o mesmo tipo de defeito.

Esse critério exige a execução de pelo menos um dado de teste em cada classe. Uma classe de equivalência representa um conjunto de estados válidos ou inválidos para condições de entrada. Uma condição de entrada normalmente é um valor numérico, um intervalo de valores, um conjunto de valores relacionados ou uma condição booleana.

3.3.1.2 Análise do Valor Limite

A análise de valor limite é considerada um complemento do critério particionamento em classes de equivalência. Este critério propõe a utilização de valores das fronteiras das classes de equivalência para os casos de teste, ao invés da utilização de qualquer valor da classe. Isso se deve ao fato de que nas fronteiras concentra-se um grande número de defeitos [24, 70]. Segundo Myers [64], este critério também pode derivar casos de teste do domínio de saída. São exigidos casos de teste que produzam resultados nos limites das classes de saída particionadas.

3.3.1.3 Grafo de Causa-Efeito

Tanto o critério particionamento em classes de equivalência quanto o critério análise de valor limite não exploram a combinação de elementos de entrada. Os requisitos de teste estabelecidos pelo critério grafo de causa-efeito são baseados nas possíveis combinações de entrada [24]. A combinação de elementos não é uma tarefa fácil porque o número de combinações pode ser inaceitável, do ponto de vista prático.

Segundo Pressman [70], esse critério fornece uma representação concisa das condições lógicas e das ações correspondentes, conforme os quatro passos seguintes:

- As condições de entrada (causas) são relacionadas às possíveis ações (efeitos) do programa;
- Um grafo de causa-efeito é construído;
- O grafo é convertido numa tabela de decisão;
- A partir das regras da tabela de decisão são derivados os casos de teste.

3.3.2 Técnica Estrutural

A técnica estrutural é baseada no conhecimento da estrutura interna da implementação. Essa técnica apresenta desvantagens resultantes de limitações levantadas por vários autores [24, 60], e é vista como complementar à técnica funcional, visto que cobre classes distintas de defeitos [60, 64, 70]. Contudo, os resultados obtidos com a aplicação da técnica estrutural têm sido considerados relevantes para as atividades de manutenção, depuração e confiabilidade de software.

A maioria dos critérios dessa técnica utiliza uma representação de programa conhecida como grafo de fluxo de controle ou grafo de programa. Um programa P pode ser decomposto em um conjunto de blocos disjuntos de comandos; a execução do primeiro comando de um bloco acarreta a execução de todos os outros comandos desse bloco, na ordem dada. Todos os comandos de um bloco, possivelmente com exceção do primeiro,

têm um único predecessor e exatamente um único sucessor, exceto possivelmente o último comando [24, 60, 73].

A Figura 3.2 apresenta o código-fonte do programa *compress.c* que será utilizado para exemplificação de conceitos do teste estrutural e baseado em defeitos. Também são apresentados os blocos de comandos que foram utilizados para criação do grafo de fluxo de controle da função *compress*. Esses blocos foram representados em forma numérica à esquerda de cada comando.

Um programa P é representado por um grafo de fluxo de controle $G = (N, A, e, s)$ onde N é o conjunto de nós, A é o conjunto de arcos (ou arestas), e é o único nó de entrada e s é o único nó de saída. Todo grafo de programa é um grafo dirigido conexo. Um nó $n \in N$ representa uma instrução simples (comando) ou uma seqüência de instruções executadas como um bloco de comandos. Cada arco $a \in A$ é um par ordenado (n, m) de nós de N que representa uma possível transferência de controle do nó n para o nó m [24, 60].

Utilizando a definição de grafo de fluxo de controle apresentada e dos blocos de comandos apresentados no código-fonte, foi gerado o grafo de fluxo de controle da função *compress* do programa, o qual é apresentado na Figura 3.3 e pode ser representado por $G = (N, A, 1, 16)$, onde $N = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$ e $A = \{(1,2), (2,3), (2,16), (3,4), (3,8), (4,5), (4,6), (5,7), (6,7), (7,15), (8,9), (8,10), (9,14), (10,11), (10,12), (11,13), (12,13), (13,14), (14,15), (15,2)\}$.

Um *caminho* de um programa é representado por uma seqüência finita de nós (n_1, n_2,..., n_k), $k \geq 2$, tal que, para todo nó n_i, $1 \leq i \leq k-1$, existe um arco (n_i, $n_i + 1$) para $i = 1, 2, ..., k-1$. Um caminho é um *caminho simples* se todos os nós que compõem esse caminho, exceto possivelmente o primeiro e o último, são distintos; se todos os nós são distintos diz-se que esse caminho é um *caminho livre de laço*. Um *caminho completo* é um caminho cujo nó inicial é o nó de entrada e cujo nó final é o nó de saída [24, 60].

Levando em consideração a função *compress*, $(2, 3, 4, 5, 7, 15, 2)$ é um caminho simples, $(2, 3, 4, 5, 7, 15)$ é um caminho livre de laços e o caminho $(1, 2, 3, 4, 5, 7, 15, 2, 16)$ é um caminho completo. Já o caminho $(7, 15, 2, 3, 4)$ é não executável e qualquer caminho completo que o inclua é também não executável, ou seja, não existe um dado de entrada

```
/*****************************************************************************
ESPECIFICAÇÃO: O programa encurta uma string padronizando a repetição de caracteres,
substitui uma sequência de 4 ou mais caracteres por ~nx, onde n é a letra A para uma
repetição de x, B para duas, e assim por diante. Grupos maiores que 26 sao quebrados
em 2. A função putrep imprime a representação de n caracteres c
*****************************************************************************/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main()
/*1*/{
/*1*/  compress();
}

void putrep(n,c)
/*1*/int n;
/*1*/int c;
/*1*/{
/*1*/  int i;
/*2*/  while((n>=4)||((c=='~')&&(n>0)))
/*3*/  {
/*3*/    putchar('~');
/*3*/    if (n < 26)
/*4*/       putchar(n-1+'A');
/*5*/    else
/*5*/       putchar(26-1+'A');
/*6*/    putchar(c);
/*6*/    n = n - 26;
/*7*/  }
/*8*/  for (i=n; i>0; i--)
/*9*/     putchar(c);
/*10*/}
```

```
void compress()
/*1*/{
/*1*/ int n,lastc,c;
/*1*/ n = 1;
/*1*/ lastc = getchar();
/*2*/ while (lastc != -1)
/*3*/ {
/*3*/   if ((c=getchar()) == -1)
/*4*/   {
/*4*/       if ((n>1)||(lastc=='~'))
/*5*/         putrep(n,lastc);
/*6*/       else
/*6*/         putchar(lastc);
/*7*/   }
/*8*/   else
/*8*/   {
/*8*/       if (c==lastc)
/*9*/           n++;
/*10*/      else if ((n>1)||(lastc=='~'))
/*11*/      {
/*11*/          putrep(n,lastc);
/*11*/          n = 1;
/*11*/      }
/*12*/      else
/*12*/      {
/*12*/          putchar(lastc);
/*13*/      }
/*14*/   }
/*15*/   lastc = c;
/*15*/ }
/*16*/}
```

Figura 3.2: Código-fonte do programa *compress.c*

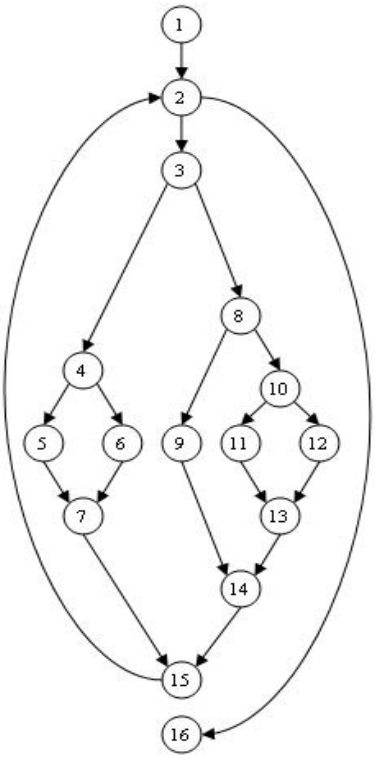


Figura 3.3: Grafo de fluxo de controle da função *compress*

que leve a sua execução.

Os critérios de teste estrutural baseiam-se em diferentes tipos de conceitos e componentes de programas para determinar os requisitos de teste e, geralmente, são classificados em critérios baseados em fluxo de controle, baseados em fluxo de dados e baseados em complexidade.

3.3.2.1 Critérios Baseados em Fluxo de Controle

Esses critérios utilizam somente características de controle da execução do programa, como comandos ou desvios, para determinar quais estruturas são necessárias. Os critérios mais conhecidos são [60]:

- critério *todos-nós* – exige que a execução do programa passe, ao menos uma vez, em cada vértice do grafo, ou seja, que cada comando do programa seja executado pelo menos uma vez;
- critério *todos-arcos* – requer que cada aresta do grafo, ou seja, cada desvio de fluxo de controle do programa, seja exercitada pelo menos uma vez; e
- critério *todos-caminhos* – requer que todos os caminhos possíveis do programa sejam executados.

Os critérios *todos-nós* e *todos-arcos* são considerados pouco exigentes, visto que a cobertura desses critérios é atingida facilmente com a execução de poucos casos de teste. Já o critério *todos-caminhos* é impraticável na maioria das vezes, especialmente quando existe a presença de laços no programa. Sendo assim, existe uma lacuna muito grande entre os critérios *todos-arcos* e *todos-caminhos* [72]. Para preencher a lacuna observada entre os critérios *todos-arcos* e *todos-caminhos* foram propostos diversos critérios baseados em fluxo de dados [60, 72, 73].

3.3.2.2 Critérios Baseados em Fluxo de Dados

A análise estática do programa fornece informações sobre ações executadas a respeito das variáveis do programa e efeitos dessas ações nos vários pontos do programa. Geralmente,

uma variável pode sofrer as seguintes ações no programa: (d) definição, (i) indefinição ou (u) uso. Essas ações são descritas detalhadamente a seguir conforme [60]:

Uma *definição de variável* ocorre quando um valor é armazenado em uma posição de memória. Uma ocorrência de variável é uma definição se ela está: i) no lado esquerdo de um comando de atribuição; ii) em um comando de entrada; ou iii) em chamadas de procedimentos como parâmetro de saída. Como exemplo para a função *compress* têm-se o conjunto de definições para o nó 1, *def(1)* = {*n, lastc*}. A definição da variável *n* consiste na atribuição (*n = 1;*) e a definição da variável *lastc* consiste na atribuição (*lastc = getchar();*). Já uma *indefinição de variável* ocorre quando, ou não se tem acesso ao seu valor, ou sua localização deixa de estar definida na memória.

Em contrapartida, o *uso de uma variável* se dá quando a referência a esta variável não a estiver definindo em um comando executável. Ou seja, há uma recuperação de um valor em uma posição de memória associada a esta variável. Dois tipos de usos são distinguidos: *c-uso* e *p-uso*. O *c-uso* (uso computacional) afeta, diretamente, uma computação que está sendo realizada ou permite observar o valor de uma variável que tenha sido definida anteriormente. Nestes casos o uso está associado a um nó do grafo de fluxo de controle. O *p-uso* (uso predicativo) afeta, diretamente, o fluxo de controle do programa. Este uso está associado a um arco do grafo. Exemplos de c-uso e p-uso para a função *compress* são apresentados a seguir juntamente com a definição de associações.

Um caminho $(i, n_1, ..., n_m, j)$, $m \geq 0$ com uma definição da variável x no nó i e que não contenha nenhuma redefinição de x nos nós $n_1, ..., n_m$ é chamado de *caminho livre de definição* com respeito a (c.r.a) x do nó i ao nó j e do nó i ao arco (n_m, j). Neste caso, pode existir uma redefinição de x no nó j [60].

Um nó i possui uma *definição global* de uma variável x se ocorre uma definição de x no nó i e existe um caminho livre de definição de i para algum nó ou para algum arco que contém um c-uso ou um p-uso, respectivamente, da variável x. Um c-uso da variável x em um nó j é um *c-uso global* se não existir uma definição de x no mesmo nó j precedendo este c-uso; caso contrário, é um *c-uso local* [60].

Rapps e Weyuker [72, 73] introduziram diversos conceitos e definições para estabelecer

a Família de Critérios de Fluxo de Dados (FCFD). Um deles foi o *grafo def-uso* (*def-use graph*) que consiste em uma extensão do grafo de fluxo de controle, na qual foram incorporadas informações semânticas do programa. O grafo def-uso é obtido a partir do grafo de fluxo de controle associando-se a cada nó i os conjuntos *c-uso(i)* = variáveis com c-uso global no nó i e *def(i)* = variáveis com definições globais no nó i, e a cada arco (i, j) o conjunto *p-uso(i, j)* = variáveis com p-usos no arco (i, j). Na Figura 3.4 é apresentado um exemplo de grafo def-uso para a função *compress* apresentada anteriormente.

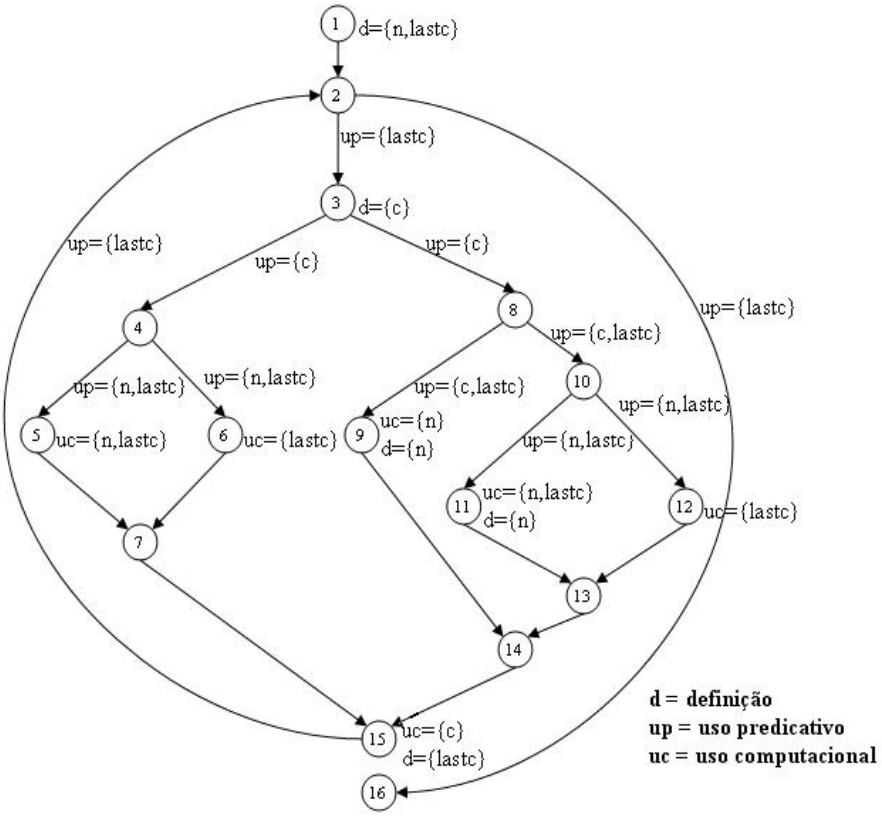


Figura 3.4: Grafo def-uso da função *compress*

Dois conjuntos foram definidos: *dcu(x,i)* = {nós j tal que $x \in$ c-uso (j) e existe um caminho livre de definição c.r.a x do nó i para o nó j} e *dpu(x,i)* = {arcos (j, k) tal que $x \in$ p-uso (j, k) e existe um caminho livre de definição c.r.a. x do nó i para o arco (j, k)}. Além disso, foram definidos os elementos requeridos de um programa: *du-caminho*, *associação-c-uso*, *associação-p-uso* e *associação* [60].

Um caminho (n_1, n_2,..., n_k) é um *du-caminho* c.r.a variável x se n_1 tiver uma definição global de x e: (1) ou n_k tem um c-uso de x e (n_1, n_2,..., n_j, n_k) é um caminho simples livre

de definição c.r.a x; ou (2) (n_j, n_k) tem um p-uso de x e (n_1, n_2,..., n_j, n_k) é um caminho livre de definição c.r.a x e $n_1, n_2, ..., n_j$ é um caminho livre de laço. Como exemplo para a função *compress* têm-se o du-caminho $(1, 2, 3, 4, 5)$ c.r.a n, visto que a variável n foi definida no nó 1 e utilizada (c-uso) no nó 5 e que o caminho exposto é simples e livre de definição c.r.a n.

Uma *associação definição-c-uso* é uma tripla (i, j, x) onde i é um nó que contém uma definição global de x e $j \in dcu(x, i)$. Como exemplo para a função *compress* tem-se a associação definição-c-uso $(1, 5, n)$. Uma *associação definição-p-uso* é uma tripla (i, (j, k), x) onde i é um nó que contém uma definição global de x e $(j, k) \in dpu(x, i)$. Como exemplo para a função *compress* tem-se a associação definição-p-uso $(1, (4, 5), n)$. Uma *associação* é uma *associação definição-c-uso*, uma *associação definição-p-uso* ou um *du-caminho* [72, 73].

A FCFD inclui os seguintes critérios:

1. Todas-definições: Requer que cada definição de uma variável x seja exercitada pelo menos uma vez, associando-a por pelo menos um c-uso ou um p-uso, através de um caminho livre de definição c.r.a x do nó onde ocorre a definição ao nó ou arco onde ocorre um uso.

2. Todos-usos: Requer que todas as associações entre cada definição de variável e subseqüentes c-usos e p-usos dessa definição sejam exercitadas pelo menos uma vez.

3. Todos-p-usos: Requer que todas as associações entre cada definição de variável e subseqüentes p-usos dessa definição sejam exercitadas pelo menos uma vez.

4. Todos-c-usos: Requer que todas as associações entre cada definição de variável e subseqüentes c-usos dessa definição sejam exercitadas pelo menos uma vez.

5. Todos-p-usos/algum-c-uso: Requer que todas as associações entre cada definição de variável e subseqüentes p-usos sejam exercitadas. Caso não haja p-uso, que pelo menos um c-uso associado a essa definição seja exercitado.

6. Todos-c-usos/algum-p-uso: Requer que todas as associações entre cada definição de variável e subseqüentes c-usos sejam exercitadas. Caso não haja c-uso, que pelo menos um p-uso associado a essa definição seja exercitado.

7. Todos-du-caminhos: Requer que todas as associações entre cada definição de variável e subseqüentes p-usos ou c-usos dessa definição sejam exercitadas por todos os caminhos livres de definição e livres de laços que cubram essas associações. Ou seja, requer que todos-du-caminhos sejam exercitados.

Maldonado et. al propuseram a Família de Critérios Potenciais Usos (FCPU) [60], os critérios dessa família não exigem a ocorrência explícita de um uso de variável. Esta é uma pequena, mas fundamental, modificação nos conceitos apresentados por Rapps e Weyuker, os quais exigem a ocorrência explícita de um uso de variável. Os elementos requeridos pela FCPU são decorrentes dos caminhos livres de definição alcançados para cada variável definida no programa. Não importa o uso real desta variável mas os pontos onde pode existir um uso. Se uma variável x é definida em um nó i e pode existir um uso em um nó j ou em um arco (j, k), então existe uma potencial associação com respeito à variável x.

As seguintes definições são introduzidas pela FCPU:

- *defg(i)* = {variável $x|x$ é definida no nó i};

- *pdcu(x,i)* = {nós $j \in N|$ existe um caminho livre de definição c.r.a x do nó i para o nó j e $x \in defg(i)$};

- *pdpu(x,i)* = {arcos $(j, k)|$ existe um caminho livre de definição c.r.a x do nó i para o arco (j, k) e $x \in defg(i)$};

- *potencial-du-caminho* c.r.a x é um caminho livre de definição $(n_1, n_2, ..., n_j, n_k)$ c.r.a x do nó n_1 para o nó n_k e para o arco (n_j, n_k), onde o caminho $(n_1, n_2, ..., n_j)$ é um caminho livre de laço e no nó n_1 ocorre uma definição de x;

- *associação potencial-definição-c-uso* é uma tripla $[i, j, x]$ onde $x \in defg(i)$ e $j \in pdcu(x, i)$;

- *associação potencial-definição-p-uso* é uma tripla $[i, (j, k), x]$ onde $x \in defg(i)$ e $(j, k) \in pdpu(x, i)$;

A seguir são apresentados exemplos relativos à função *compress* para as definições propostas:

- Associação potencial-definição-c-uso: $[1, 7, n]$.
- Associação potencial-definição-p-uso: $[1, (8, 10), n]$.
- Potencial-du-caminho: $(1, 2, 3, 4, 5, 7, 15)$.

Os critérios que compõem a FCPU propostos por Maldonado, Chaim e Jino são:

1. *todos-potenciais-usos* - PU: para todo nó $i \in G$ tal que $defg(i) \neq \emptyset$ (onde $\emptyset$ é o conjunto vazio) e para toda variável x tal que $x \in defg(i)$ requer que todas as *associações potenciais-definição-p-uso* e todas as *associações potenciais-definição-c-uso* sejam exercitadas pelo menos uma vez.

2. *todos-potenciais-du-caminhos* - PDU: para todo nó $i \in G$ tal que $defg(i) \neq \emptyset$ e para toda variável x tal que $x \in defg(i)$ requer que todos os *potenciais du-caminho* c.r.a x sejam exercitados pelo menos uma vez.

3. *todos-potenciais-usos/DU* - $PUDU$: para todo nó $i \in G$ tal que $defg(i) \neq \emptyset$ e para toda variável x tal que $x \in defg(i)$ requer o exercício das mesmas potenciais associações de fluxo de dados c.r.a x, mas colocando a restrição de que tal associação deverá ser exercitada por um du-caminho, a partir do ponto onde ocorreu a definição até o ponto onde poderá ocorrer um possível uso.

Um limitação do teste estrutural é relativa a caminhos não executáveis. Um caminho pode ser não executável e qualquer caminho completo que o inclua é também não executável, ou seja, não existe um dado de entrada que leve à sua execução. Sendo assim, é preciso a intervenção do testador para determinar quais são os caminhos não executáveis para um programa pois não existe um algoritmo, que dado um caminho completo qualquer, decida se o caminho é executável e forneça o conjunto de valores que causam a execução

desse caminho [30]. Um elemento requerido por um determinado critério estrutural será não executável se não existir caminho executável que o cubra.

3.3.3 Técnica Baseada em Defeitos

Esta técnica utiliza informações sobre os tipos de defeitos mais freqüentes no processo de desenvolvimento de software para derivar os requisitos de teste. A ênfase da técnica está nos erros que o programador ou projetista pode cometer durante o desenvolvimento e nas abordagens que podem ser usadas para detectar a sua ocorrência. O objetivo é mostrar a presença ou ausência de tais defeitos no programa [24]. Dois critérios típicos que se concentram em defeitos são os critérios de Semeadura de Erros (*Error Seeding*) e de Análise de Mutantes (*Mutation Analysis*) [25].

3.3.3.1 Critério Análise de Mutantes

O critério análise de mutantes baseia-se em duas hipóteses, a hipótese do programador competente, a qual assume que programadores competentes constroem programas muito próximo do correto; e o efeito de acoplamento, o qual pressupõe que defeitos complexos estão relacionados a defeitos simples e alguns estudos empíricos já confirmaram esta hipótese, mostrando que conjuntos de casos de teste capazes de revelar defeitos simples são também capazes de revelar defeitos complexos [24].

Conforme DeMillo et. al [25], partindo-se dessas hipóteses, são gerados, a partir do programa em teste, programas mutantes através de operadores de mutação. Os operadores são regras que são aplicadas para definir as alterações no programa em teste. Um dado de teste deve ser gerado para diferenciar o comportamento do programa original do comportamento de um programa mutante. Quando existir essa diferença, o mutante é dito estar morto e o programa original está livre do erro descrito por esse mutante. O objetivo é obter casos de teste que resultem apenas em mutantes mortos. Os mutantes equivalentes são dispensáveis, visto que o mutante e o programa original apresentam sempre o mesmo resultado, para qualquer dado de teste. Neste caso, tem-se um conjunto de casos de teste T adequado ao programa P em teste, no sentido de que, ou P está

correto, ou possui defeitos pouco prováveis de ocorrerem. Sendo assim, o critério análise de mutantes requer que todos os mutantes sejam mortos.

A equivalência entre programas é uma questão indecidível e requer a intervenção do testador. Contudo, essa limitação não exige o abandono do problema por não apresentar solução. Alguns métodos e heurísticas têm sido propostos para determinar a equivalência de programas em uma grande porcentagem dos casos de interesse. Um dos maiores problemas para a aplicação do critério Análise de Mutantes está relacionado ao seu alto custo, visto que o número de mutantes gerados, mesmo para pequenos programas, pode ser muito grande, exigindo um tempo de execução muito alto [24].

Os operadores de mutação são construídos para satisfazer um entre dois propósitos [24]:

- induzir mudanças sintáticas simples com base nos defeitos típicos cometidos pelos programadores (como trocar o nome de uma variável); ou
- forçar determinados objetivos de teste (como executar cada arco do programa).

Para exemplificar a geração de um programa mutante foi gerado um mutante da função *compress*, baseado no operador de mutação em constantes conforme apresentado na Figura 3.5. Esse operador trocou a constante 1 por 0 na atribuição assinalada.

3.4 Ferramentas de Teste de Software

A aplicação de critérios de teste sem o apoio de uma ferramenta de software é propensa a erros. Sendo assim, diversas ferramentas de teste foram desenvolvidas nas últimas décadas com o intuito de apoiar a aplicação de critérios de teste. Dentre elas destacam-se as ferramentas *Poke-Tool* (***Po****tential-Uses* ***Cri****teria* ***Tool*** *for program testing*) [14] e *Proteum*(***Pro****gram* ***Te****sting* ***U****sing* ***M****utants*) [21, 23]. Essas ferramentas são acionadas a partir da ferramenta desenvolvida neste trabalho por conseqüência são apresentadas de forma detalhada a seguir.

```
void compress()
{
 int n,lastc,c;
 n = 0;
 lastc = getchar();
 while (lastc != -1)
 {
   if ((c=getchar()) == -1)
   {
      if ((n>1)||(lastc=='~'))
        putrep(n,lastc);
      else
        putchar(lastc);
   }
   else
   {
       if (c==lastc)
           n++;
       else if ((n>1)||(lastc=='~'))
       {
            putrep(n,lastc);
            n = 1;
       }
       else
       {
            putchar(lastc);
       }
    }
    lastc = c;
  }
}
```

Figura 3.5: Mutante da função *compress*: operador em constantes

3.4.1 A Ferramenta de Teste *Poke-Tool*

A ferramenta *Poke-Tool* (*Potencial Uses Criteria Tool for Program Testing*) é um ambiente de teste para análise de cobertura estrutural que apóia a aplicação dos critérios Potenciais-Usos e também de outros critérios estruturais como Todos-Nós e Todos-Arcos. A versão utilizada neste trabalho apóia o teste de unidade em programas C (entendendo por unidade uma função em C). No entanto, ela é uma ferramenta multilinguagem e pode trabalhar com programas escritos em Clipper, COBOL e FORTRAN-77 [14, 15, 16].

A *Poke-tool* é uma ferramenta orientada à sessão, o termo sessão de trabalho ou de teste é utilizado para designar as atividades envolvendo um teste. O teste pode ser realizado em etapas onde são armazenados os estados intermediários da aplicação de teste a fim de que possam ser recuperados posteriormente. O conjunto de atividades a serem realizadas é composto por: fornecer o programa para teste, compilar o programa em teste, gerar um programa instrumentado, executar os casos de teste, avaliar os casos de teste e visualizar os resultados. Na versão utilizada, a execução se dá por meio de scripts, mas também há uma versão com interface gráfica e uma proposta de extensão para apoiar o

tratamento da não executabilidade de caminhos [16].

Como saída, a ferramenta fornece ao usuário o conjunto de *arcos primitivos*, o grafo def-uso obtido do programa em teste, o programa instrumentado para teste, o conjunto de associações necessárias para satisfazer o critério selecionado e o conjunto de associações ainda não exercitadas. O conjunto de arcos primitivos consiste de arcos que uma vez executados garantem a execução de todos os demais arcos do grafo de programa [16].

3.4.2 A Ferramenta de Teste *Proteum*

A ferramenta *Proteum* (*Program Testing Using Mutants*) apóia o critério Análise de Mutantes e está disponível para os sistemas operacionais SunOS, Solaris e Linux. A ferramenta pode ser utilizada via interface gráfica ou linha de comando e as funções implementadas possibilitam que alguns dos recursos sejam executados automaticamente, enquanto que para outros são fornecidas facilidades para que o testador possa realizá-los. Desse modo, a ferramenta pode ser utilizada como instrumento de avaliação bem como de seleção de casos de teste [21, 23].

Um dos pontos essenciais para a aplicação do critério Análise de Mutantes é a definição do conjunto de operadores de mutação. A *Proteum* conta com 71 operadores de mutação divididos em quatro classes: mutação de comandos (*statement mutations*), mutação de operadores (*operator mutations*), mutação de variáveis (*variable mutations*) e mutação de constantes (*constant mutations*). Esses operadores são detalhados no Apêndice A.

É possível escolher os operadores de acordo com a classe de defeitos que se deseja enfatizar, permitindo que a geração de mutantes seja feita em etapas ou até mesmo dividida entre vários testadores trabalhando independentemente. Sendo assim, cada operador de mutação é associado a uma determinada classe de defeito.

A ferramenta permite ao testador avaliar a adequação de um conjunto de casos de teste T para um determinado programa P e com o resultado dessa avaliação o testador pode melhorar o conjunto T visando satisfazer o critério Análise de Mutantes. As operações mínimas suportadas pela ferramenta são:

- Manipulação de casos de teste: execução, inclusão e exclusão.

- Manipulação de mutantes: geração, execução e análise.
- Análise de adequação: escore de mutação e relatórios estatísticos.

3.5 Considerações Finais

Este capítulo apresentou uma revisão bibliográfica referente à atividade de teste. Inicialmente, foram apresentados os conceitos básicos e uma estratégia para teste. Também foram descritas as técnicas e os critérios de teste mais utilizados nessa atividade. Por fim foram apresentadas as ferramentas *Poke-Tool* e *Proteum* que são acionadas pela ferramenta desenvolvida neste trabalho. Vale ressaltar que as técnicas de teste apresentadas são consideradas complementares por revelarem diferentes tipos de defeitos. As ferramentas apresentadas *Poke-Tool* e *Proteum* auxiliam a aplicação dos critérios dessas técnicas, entretanto, elas não fazem parte de um ambiente integrado para o teste. Cada uma delas produz diferentes informações que, apesar de relacionadas ao mesmo programa, são apresentadas e mantidas separadamente. Por isso, mesmo o usuário testador, não é capaz de perceber e utilizar essas relações, principalmente para utilização no teste de regressão, assunto do próximo capítulo. No próximo capítulo são apresentados alguns conceitos relativos ao teste de regressão, juntamente com as principais técnicas e ferramentas. Também é apresentada uma comparação das técnicas existentes e uma descrição dos trabalhos relacionados a este objeto de estudo.

CAPÍTULO 4

TESTE DE REGRESSÃO

Este capítulo apresenta alguns conceitos relacionados à manutenção de software e à atividade de teste de regressão. As técnicas de teste de regressão, pertinentes a este trabalho, são descritas juntamente com as ferramentas e estudos empíricos relacionados.

4.1 Terminologia e Conceitos Básicos

De acordo com Pressman [70], a fase de manutenção do software ocorre após a entrega do sistema ao usuário. O termo manutenção, quando aplicado à Engenharia de Software, ganha novas conotações. Esse termo pode representar a correção de defeitos de um software, ou ainda, este software pode ser expandido e/ou melhorado para atender novas necessidades do usuário. A manutenção de software pode representar mais de 50% de todo esforço despendido por uma empresa de desenvolvimento de software. E à medida que mais softwares são produzidos, a porcentagem continua a aumentar. As mudanças são inevitáveis quando sistemas computacionais são desenvolvidos, para tanto, é necessário o desenvolvimento de mecanismos para avaliar, controlar e fazer modificações [70].

A atividade de manutenção de um software ocorre devido a motivos distintos, como detecção de defeitos pelo usuário, mudanças de especificação, adaptações a novos ambientes, necessidade de implementação de novas funcionalidades, etc. Existem, basicamente, quatro categorias de modificações realizadas durante a manutenção [70]:

1. Correção: É provável que o usuário detecte defeitos no software, mesmo que este tenha sido submetido a uma rigorosa garantia de qualidade. Para tanto uma manutenção corretiva é necessária para correção dos defeitos;

2. Adaptação: Ao longo do tempo, o ambiente original (hardware, sistema operacional, entre outras características) para o qual o software foi desenvolvido é suscetível a

mudanças. A manutenção adaptativa é responsável por realizar mudanças para que o software se adapte ao ambiente externo;

3. Evolução: modificações são realizadas para adicionar novas funcionalidades ao software;

4. Prevenção: modificações são realizadas para facilitar futuras manutenções. Na sua essência, a manutenção preventiva faz mudanças em softwares para que possam ser corrigidos, adaptados e melhorados mais facilmente.

Qualquer modificação deve ser testada, independentemente da sua categoria, visto que pode efetivamente alterar a estrutura do programa. As modificações corretivas e evolutivas requerem novos casos de teste funcionais, enquanto as modificações adaptativas e preventivas não requerem necessariamente novos casos de teste, já que, eventualmente, não implicam em mudanças nos requisitos.

Tanto as funcionalidades modificadas do software quanto as funcionalidades que não sofreram modificações devem ser testadas, com a finalidade de garantir que as funcionalidades não modificadas não tenham sido afetadas pela manutenção. Esse processo de teste realizado durante a fase de manutenção do software é denominado teste de regressão [55]. Segundo Pressman [70], o teste de regressão é a reexecução de algum subconjunto de testes que já foram conduzidos para garantir que as modificações não propagaram efeitos colaterais indesejáveis.

Rothermel e Harrold [79] definem o ciclo de vida do teste de regressão em duas fases:

1. Fase Preliminar: Esta fase inicia após algumas liberações do software. Nessa fase os programadores efetuam correções de defeitos e melhorias.

2. Fase Crítica: Já esta fase inicia quando as correções e melhorias da fase anterior estão prontas. A atividade dominante da fase crítica é o teste de regressão. Nesta fase a minimização dos custos é mais importante, visto que o tempo para execução do teste de regressão geralmente é bastante limitado.

A estratégia ideal para testar um programa que sofreu manutenção consiste em executar todos os casos de teste do programa original, visando a identificação de novos defeitos introduzidos pelas modificações. A utilização de um conjunto de casos de teste já planejado minimiza os esforços da criação de novos casos de teste e também permite uma comparação direta da saída do programa modificado com a saída do programa original. No entanto, executar todos os casos de teste existentes é muito custoso. Assim, um subconjunto de casos de teste é selecionado para retestar o programa modificado. Às vezes, os casos de teste existentes não são suficientes para avaliar as modificações no software, sendo assim, novos casos de teste devem ser criados.

Sendo assim, Hartmann e Robson [39] salientam a existência de dois problemas cujas técnicas de teste de regressão buscam solucionar: os problemas da atualização e o da seleção de testes. O primeiro consiste em manter o conjunto de casos de teste T adequado mesmo após as alterações, visto que a identificação e eliminação de casos de teste irrelevantes não é uma tarefa trivial. Já o segundo problema consiste em selecionar um conjunto de casos de teste $T' \subset T$ para retestar o programa após as modificações.

4.2 Metodologias de Teste de Regressão

Considerando um programa P, P' uma versão modificada de P, e T um conjunto de casos de teste desenvolvido para P. O teste de regressão consiste em validar P'.

Para facilitar o teste de regressão, testadores geralmente reusam T, contudo novos casos de teste também são necessários para testar novas funcionalidades. Tanto a reutilização de T quanto a criação de novos casos de teste são importantes. No entanto, é o reuso de casos de teste que é relevante para este trabalho. De acordo com Rothermel et. al [77] foram consideradas quatro metodologias relacionadas ao teste de regressão e ao reuso de casos de teste: *retest-all*, seleção de casos de teste, redução de conjunto de casos de teste e priorização de casos de teste:

4.2.1 *Retest-all*

Quando o programa P é modificado, é criado P', nesse momento os testadores simplesmente reutilizam todos os casos de teste não obsoletos de T para testar P'; essa técnica é conhecida como *retest-all* [53]. (Casos de teste de T que não são aplicáveis à P' são obsoletos e precisam ser reformulados ou descartados [53].) A técnica *retest-all* representa uma prática de uso comum [66].

4.2.2 Seleção de Casos de Teste

A técnica *retest-all* pode ser cara: re-executar todos os casos de teste requer esforço humano e tempo inaceitáveis. As técnicas de seleção de casos de teste utilizam informações sobre P, P' e T para selecionar um subconjunto de T para testar P'. Diversas técnicas de seleção de casos de teste para o teste de regressão foram propostas e estas, por sua vez, utilizam diferentes métodos para realizar a seleção de casos de teste, dentre eles pode-se citar: Equações Lineares [29, 39, 40], Execução Simbólica [95], Análise de Caminhos [6], Fluxo de Dados [37, 67, 88], Cobertura dos Critérios Potenciais-Usos [33], Mutação Seletiva [61], Baseada em *Program Dependence Graph* (PDG) [4], Baseada em *System Dependence Graph* (SDG) [8], Identificação de Modificações [86], *Firewall* [54], Identificação de *Clusters* [51], *Slicing* [2], Busca em Grafo [76, 78] e Entidades Modificadas [18]. Rothermel e Harrold descrevem, detalhadamente, as técnicas de seleção de casos de teste em [79]. Estudos empíricos mostraram que algumas dessas técnicas [18, 34, 81, 82] podem ser eficazes em termos de custos.

Uma relação custo-benefício entre técnicas de seleção de casos de teste envolve segurança e eficiência. Técnicas de seleção seguras (por exemplo: [18, 80, 90]) garantem que, sob certas condições, casos de teste não selecionados não revelam defeitos em P' [79]. Obter segurança, no entanto, pode requerer a inclusão de um número maior de casos de teste do que a capacidade de execução no tempo disponível. Técnicas não seguras (por exemplo: [29, 54]) sacrificam segurança pela eficiência, selecionando casos de teste que, de certo modo, são mais úteis que os excluídos. Um caso específico de técnicas não seguras envolve técnicas que tentam minimizar o conjunto de casos de teste selecionado em relação

a um conjunto fixo de requisitos de cobertura e informações de mudanças, buscar o menor custo de execução de teste possível e consistente com a cobertura das partes modificadas do código [29, 39].

Uma maneira padrão para projetar um conjunto mínimo de testes de regressão é identificar classes de equivalência para cada entrada e então utilizar somente um valor limite de cada classe. Contudo, esse padrão de projeto faz com que a quantidade de casos de teste cresça exponencialmente em relação à quantidade de entradas do software. Além disso, a identificação de classes de equivalência é subjetiva e inexata e os limites podem mudar ao longo do tempo.

4.2.3 Redução de Conjunto de Casos de Teste

Com a evolução de P, novos casos de teste podem ser incluídos em T para validar as novas funcionalidades. Ao longo do tempo, T aumenta, e seus casos de teste tornam-se redundantes em termos de código ou funcionalidades exercitadas. Técnicas de redução de conjunto de casos de teste [17, 36, 44, 65] abordam este problema utilizando informações sobre P e T para remover permanentemente os casos de teste redundantes de T, possibilitando um reuso de T mais eficiente. A redução de conjunto de casos de teste difere da seleção de casos de teste, visto que a última não remove permanentemente os casos de teste de P, mas simplesmente copia estes casos de teste para utilizar em uma versão específica P' de P, retendo casos de teste não utilizados para utilizar em versões futuras.

A redução de conjunto de casos de teste reduz o custo da execução através da redução do tamanho do conjunto de casos de teste, validando e gerenciando conjuntos de casos de teste sobre versões futuras do software. Um possível inconveniente da redução de conjunto de casos de teste, no entanto, é que a remoção de casos de teste do conjunto pode prejudicar a capacidade de detecção de defeitos. Enquanto alguns estudos [94] mostraram que a redução de um conjunto de casos de teste pode produzir ganhos substanciais para eficácia de detecção de defeitos e com baixo custo; Outros estudos [82] mostraram que a redução de casos de teste pode reduzir, significativamente, o poder de detecção de defeitos de um conjunto.

4.2.4 Priorização de Casos de Teste

As técnicas de priorização de casos de teste [27, 44, 83, 87, 94], organizam os casos de teste de forma que aqueles que possuem prioridade maior, de acordo com algum critério, são executados antes que os que possuem prioridade menor durante o processo de execução do teste de regressão. Por exemplo, testadores poderiam desejar que os casos de teste fossem executados em uma ordem que obtivesse cobertura de código o mais rápido possível, que exercitasse funcionalidades em ordem de freqüência de uso, ou ainda, aumentar a probabilidade de detecção de defeitos no início dos testes.

Resultados empíricos [27, 83, 94] sugerem que várias técnicas simples de priorização podem melhorar significativamente um dos objetivos de performance do teste: a taxa com que o conjunto de casos de teste detecta defeitos. Uma taxa de detecção de defeitos melhorada durante o teste de regressão provê *feedback* antecipado em relação à qualidade do programa em teste e permite antecipar a correção dos defeitos pelos desenvolvedores. Esses resultados também sugerem, no entanto, que a relação custo-benefício das técnicas de priorização variam de acordo com o tamanho de: programas, conjuntos de casos de teste e tipos de mudanças.

Diversas técnicas de priorização têm sido propostas [27, 44, 83, 87, 94]. Contudo, as técnicas que prevalecem na literatura e na prática envolvem informações simples de cobertura de código, e às vezes complementam tais informações de cobertura com detalhes de onde o código foi alterado. A abordagem descrita em [87] mostrou-se extremamente eficiente em grandes sistemas da *Microsoft*, contudo, tem sido demonstrado que a eficácia das abordagens varia de acordo com vários fatores, incluindo características do conjunto de casos de teste utilizado.

4.2.4.1 Priorização de Casos de Teste Baseada em Fluxo de Dados

Rummel et al. [84] propuseram uma técnica de priorização de testes de regressão focada em todas as definições e usos de variáveis. É realizada a análise intraprocedural do fluxo

de dados para identificar cada associação de definição e uso de variáveis de cada método. Uma vez que o conjunto de casos de teste foi executado, pode ser calculada a adequação de cada caso de teste. Então é utilizada uma função para identificar o conjunto de métodos que foi testado por um caso de teste. A partir daí uma equação define a adequação cumulativa de um caso de teste com relação ao número de requisitos de teste cobertos e o número total de requisitos de teste para todos os métodos testados. A partir do resultado dessa equação de adequação os casos de teste são priorizados. O trabalho de Rummel et al. apresenta como contribuições: a descrição de uma técnica que utiliza informações de fluxo de dados para priorizar testes de regressão; a descrição formal dos algoritmos e equações que são usadas para instrumentar o programa testado, monitorando cobertura durante a execução do conjunto de testes e calculando a adequação dos testes; e a avaliação de utilização de tempo e espaço durante a execução da ferramenta e avaliação da efetividade da priorização dos testes.

4.3 Aplicação de Aprendizado de Máquina no Teste de Regressão

Na literatura, muitos trabalhos utilizam técnicas de Aprendizado de Máquina (AM) (Capítulo 2) para apoiar o teste de regressão. Essas técnicas são utilizadas para encontrar relacionamentos entre os conjuntos de dados oriundos das atividades de teste de software. Têm-se como exemplos desses dados: o domínio de entrada do programa, os caminhos percorridos pelos casos de teste, defeitos revelados pelos casos de teste, entre outros. Esses relacionamentos são utilizados para seleção, priorização e redução do conjunto de dados de teste para o teste de regressão. A seguir são apresentados alguns trabalhos relacionados, baseados em técnicas de AM para apoio ao teste de regressão.

4.3.1 Seleção de Casos de Teste de Regressão Utilizando Redes *Info-Fuzzy* [52]

O estudo realizado por Last et. al [52] teve como objetivo utilizar modelos induzidos de mineração de dados para teste de software. Estes podem ser utilizados para recuperar especificações incompletas ou inexistentes, que, por sua vez, servem para projetar um conjunto mínimo de testes de regressão e avaliar se as saídas do sistema estão corretas ao testar novas versões potencialmente defeituosas. O estudo consistiu em aplicar um algoritmo de mineração de dados chamado *Info-Fuzzy Network* (IFN). Este algoritmo utiliza como entrada um conjunto de casos de teste T, gerado aleatoriamente, a partir das especificações de entradas e saídas do programa original e os dados da execução do conjunto T sobre o programa original.

A rede *info-fuzzy* é como uma estrutura de árvore sem inteligência conforme exposto na Figura 4.1, onde o mesmo atributo de entrada é utilizado através de todos os nós de um nível. Os componentes da rede incluem um nó raíz, um número variável de camadas ocultas (uma camada para cada entrada selecionada), e a última camada representando os valores de saída. Cada nó da última camada é associado à uma classe de equivalência do domínio de um atributo. Visto que o modelo proposto é capaz de prever os valores contínuos de um atributo de entrada, os nós da última camada da IFN representam subintervalos disjuntos de um intervalo contínuo. Além disso, não há limite de nós de saída para uma IFN.

O procedimento de indução da IFN é um algoritmo *greedy* que nem sempre encontra uma solução ótima para a ordenação dos atributos de entrada. Embora algumas funcionalidades dependam altamente da ordem dos atributos de entrada, ordens alternativas são aceitáveis na maioria dos casos. Os modelos de IFN precisam ter um nível de exatidão elevado para as previsões realizadas, contudo um nível ótimo é irreal. Por outro lado, o fato desses modelos serem compactos pode ajudar a recuperar os requisitos dominantes dos dados de execução e, conseqüentemente, construir um conjunto de casos de teste compacto.

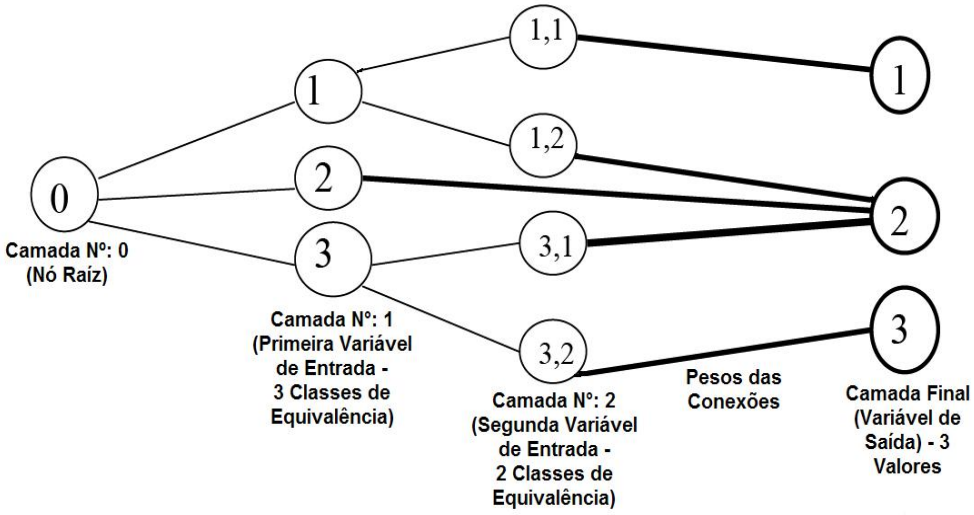


Figura 4.1: Estrutura de árvore da Rede *Info-Fuzzy* adaptada de [52]

Outro fator importante é a estabilidade do algoritmo em relação aos dados de treinamento, desde que o algoritmo seja treinado com um conjunto pequeno e aleatório de valores de entrada gerados. Na Figura 4.2, é demonstrada a arquitetura do ambiente baseado em IFN.

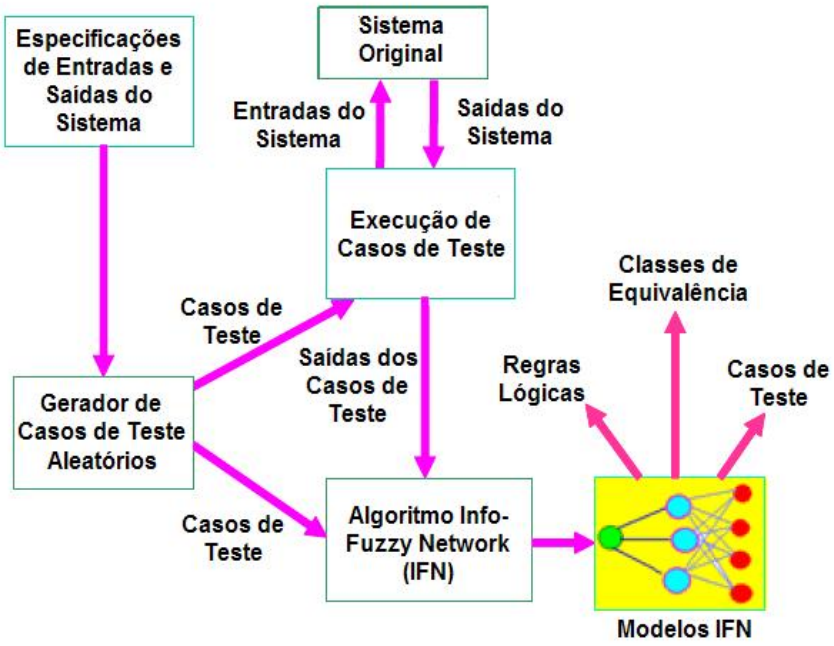


Figura 4.2: Arquitetura do ambiente baseado em IFN adaptada de [52]

O gerador aleatório de casos de teste obtém uma lista de entradas e saídas com seus tipos (discretos, contínuos, etc.) da especificação do sistema. Nenhuma informação sobre

os requisitos funcionais é necessária, desde que o algoritmo de IFN revele, automaticamente, os relacionamentos de entrada e saída dos casos de teste de treinamento aleatórios que foram gerados. O algoritmo de IFN é treinado através das entradas fornecidas pelo gerador aleatório de casos de teste e das saídas dos casos de teste obtidas do sistema original por meio do módulo Execução de Casos de Teste. A seguir segue uma breve descrição de cada um dos módulos da Figura 4.2:

1. Sistema Original: Este módulo representa um programa em uma versão anterior à modificação. Este módulo deve ter uma relação bem definida entre as entradas e saídas retornadas.

2. Especificações de Entradas e Saídas do Sistema: Dados básicos para cada variável de entrada e saída do sistema original. Inclui nomes de variáveis, tipos e lista ou intervalo de valores possíveis.

3. Gerador de Casos de Teste Aleatórios: Esse módulo gera combinações aleatórias de valores do intervalo para cada variável de entrada. Esses casos de teste de treinamento gerados são utilizados pelo algoritmo de IFN e pelo módulo de execução de casos de teste.

4. Execução de Casos de Teste: Esse módulo alimenta o sistema original com os casos de teste de treinamento gerados pelo gerador de casos de teste aleatórios. Essa execução pode ser feita com ferramentas comerciais de automatização de testes. Sendo assim, esse módulo obtém as saídas de cada caso de teste executado e as envia ao algoritmo IFN.

5. Algoritmo *Info-Fuzzy Network* (IFN): As entradas desse algoritmo são os casos de teste de treinamento gerados pelo gerador de casos de teste aleatórios e as saídas produzidas para cada caso de teste executado pelo módulo de execução de casos de teste. O algoritmo também usa as descrições das variáveis. O algoritmo é executado repetidamente para encontrar um subconjunto das variáveis de entrada relevantes a cada saída e ao conjunto correspondente de casos de teste não redundantes. O

conjunto de casos de teste T' é gerado a partir das classes de equivalência geradas automaticamente.

O estudo exposto apresentou e avaliou uma metodologia para automatização da análise de entradas e saídas de um sistema. O método produz automaticamente um conjunto de casos de teste não redundantes cobrindo a maioria das relações funcionais comuns existentes no software (incluindo as classes de equivalência correspondentes).

4.3.2 Seleção de Casos de Teste de Regressão Utilizando Redes Neurais [85]

Saraph et. al [85] propuseram uma metodologia para identificar os casos de teste mais importantes automaticamente. Estes casos de teste envolvem atributos de entrada que afetam o valor de uma saída. Sendo assim, reduz-se o número de casos de testes relacionando entradas e saídas. Como resultado dessa metodologia têm-se uma lista ordenada de funcionalidades e classes de equivalência para atributos de entrada de um dado programa. A metodologia elaborada é baseada em Redes Neurais, dando continuidade ao estudo realizado com Redes *Info-Fuzzy* [52], descrito na seção anterior.

Para sistemas que possuem muitos atributos de entrada e de saída, o número de combinações de casos de teste funcionais é muito elevado, sendo impossível listar e executar todos devido à limitação de recursos. Uma solução é estabelecer um critério para escolha dos casos de teste mais efetivos, descartando os demais. Contudo, têm-se inúmeras características de efetividade e importância para escolha de casos de teste, as quais dependem da aplicação em questão, da prática de teste adotada e dos recursos disponíveis para teste. Sendo assim, a redução de casos de teste é um tópico que merece atenção.

Análise de Relações Entrada-Saída é um tipo de teste caixa preta que identifica atributos de entrada que afetam o valor de uma saída específica. Esta prática é concentrada no relacionamento entre entradas e saídas. Na perspectiva do teste funcional, identificar tais relações é a primeira etapa para redução de um conjunto de casos de teste. A geração de casos de teste pode ser automatizada caso as relações entradas-saídas sejam identifica-

das, uma maneira de identificar essas relações foi descrita em [52] utilizando IFN. Já este estudo continua o trabalho anterior, mas utilizando redes neurais ao invés de IFN, visto que geram resultados melhores.

Uma Rede Neural treinada pode ser utilizada como um oráculo. Para tanto, uma variedade de cortes e de algoritmos de extração de regras podem ser utilizadas para realizar o treinamento. A adequação do treinamento de redes neurais, corte e extração de regras para mineração de dados são discutidos em [58]. A metodologia proposta pode ser dividida em quatro fases distintas conforme apresentado na Figura 4.3. Cada uma dessas fases apresentadas na figura é detalhada como segue:

Figura 4.3: Arquitetura do ambiente baseado em Redes Neurais adaptada de [85]

1. Criação e treinamento da rede: Foi treinada e alimentada uma rede com três camadas através do método (*back-propagation*). O algoritmo tenta encontrar o melhor conjunto de pesos para um dado conjunto de exemplos pela minimização do valor da função de erro para todos os exemplos de treinamento. Os dados de treinamento são gerados aleatoriamente criando os atributos de entrada, que, por sua vez, são utilizados para alimentar o programa original P. Assim, são armazenadas as saídas geradas que são associadas aos atributos de entrada selecionados.

2. Corte: Um algoritmo de corte pode ser utilizado em uma rede neural direta (*feed-forward*) que possui 3 camadas. O objetivo desse algoritmo é cortar o maior número possível de conexões, deixando somente as ligações mais importantes. O termo de penalidade é uma parte da função de erro e identifica conexões desnecessárias pela associação de pesos baixos para elas. O objetivo é remover o maior número de

conexões, contanto que permaneça em um nível aceitável de exatidão. Os métodos de ordenação de funcionalidades podem ser considerados como um produto das fases de treinamento e corte. Os atributos mais significativos, apresentados pela ordenação de funcionalidades, podem ser considerados para construção dos casos de teste. Os casos de teste incluem os atributos com pontuação maior, já os atributos com pontuação menor são eliminados.

3. Ordenação de funcionalidades: A fase de treinamento utiliza uma função de penalidade que é parte da função objetivo, a qual atribui valores mais elevados aos pesos mais importantes. Após o treinamento, os pesos são associados às conexões das camadas ocultas com a camada de saída e as conexões da camada de entrada com as camadas ocultas indicando quais conexões serão mantidas e quais serão cortadas através da análise do valor dos pesos. Essa informação é utilizada para ordenar os atributos de entrada no primeiro método de ordenação de funcionalidades. Ordenar as entradas de acordo com o produto de pesos (produto dos pesos da camada de entrada para as camadas ocultas por seus correspondentes pesos das camadas ocultas para a camada de saída) e ordenar os atributos depois da fase de treinamento é o método mais intuitivo para ordenação de funcionalidades. Durante o corte, os pesos mudam constantemente necessitando o retreinamento da rede.

4. Extração de regras: O objetivo dessa fase é expressar as relações Entrada-Saída preservadas pelo corte na forma de regras *Se-Então* para extrair informações de classes de equivalência. Depois da fase de corte, uma possível ponte entre entradas e saídas da rede cortada são os valores unitários de ativação ocultos. Como esses valores são contínuos, eles precisam ser discretizados antes de extrair as regras da rede. Para tanto, um algoritmo de (*clustering*) é utilizado. Já o algoritmo de extração de regras é executado em duas partes, a primeira utiliza as saídas geradas com as combinações possíveis dos valores discretizados. Uma vez que as saídas dos valores unitários de ativação ocultos são conhecidas, as entradas que podem gerar os valores de ativação discretizados precisam ser identificadas em ordem para ligar

as entradas às saídas.

5. Geração de casos de teste: Os casos de teste são gerados após o corte e a extração de regras. Após o término da fase de corte, os valores possíveis dos atributos são utilizados como classes de equivalência para construir casos de teste. As definições de classes de equivalência extraídas antes da fase de extração de regras são utilizadas para geração de casos de teste.

A metodologia foi aplicada a um sistema de múltiplas saídas, na qual a maioria das entradas que afetam as saídas da aplicação foram identificadas. A identificação dos relacionamentos de entradas e saídas resultou numa redução considerável no número de casos de teste se comparado ao teste exaustivo. O método proposto é uma prática de teste caixa preta onde nenhuma informação de especificação ou código fonte é necessária para sua execução. Outros benefícios da metodologia são a determinação das classes de equivalência para atributos nominais e a obtenção de uma lista ordenada de funcionalidades.

4.3.3 Priorização de Casos de Teste de Regressão Utilizando Algoritmos de Busca [56]

Outro estudo relacionado refere-se à utilização de algoritmos de busca para priorização de casos de teste de regressão. Abordagens de priorização foram propostas com o intuito de executar os testes mais importantes primeiro tornando o teste de regressão mais efetivo. Trabalhos anteriores sobre a priorização de casos de teste de regressão foram desenvolvidos utilizando algoritmos *Greedy*. Mas esses algoritmos produzem uma solução sub-ótima, diferentemente de algoritmos de busca evolucionários e de metaheurística que resolvem esse problema.

O trabalho apresentado por Harman e Hierons [56] focaliza em técnicas de priorização de casos do teste para a cobertura de código, incluindo cobertura de blocos de código, cobertura de decisões e cobertura de comandos. Foram estudadas duas técnicas de busca metaheurística: *Hill Climbing* e Algoritmos Genéticos e três algoritmos *Greedy*: *Greedy*, *Additional Greedy* e *2-Optimal Greedy*. Desses, somente *Greedy* e *Additional Greedy* ha-

viam sido estudados anteriormente nesse contexto. Tais técnicas são brevemente descritas no Capítulo 2.

Os resultados obtidos referente ao estudo da aplicação dessas cinco técnicas sobre seis programas de teste foram:

1. Os algoritmos *Additional Greedy e 2-Optimal Greedy* apresentam os melhores resultados. Contudo, a diferença de desempenho desses para o algoritmo Genético não é significativa.

2. Os algoritmos *Additional Greedy, 2-Optimal Greedy* e Genético sempre obtém performance superior ao algoritmo *Greedy*. Estes resultados são encontrados para todos os programas estudados.

3. Os resultados para o algoritmo *Hill Climbing* revelam que o domínio de aptidão do espaço de busca envolvido é multimodal.

4.3.4 Refinamento de Casos de Teste Utilizando AM [11]

Briand et. al [11] propuseram uma metodologia baseada em AM para analisar os pontos fracos de um conjunto de casos de teste e melhorá-lo iterativamente. Esse estudo foi chamado de re-engenharia de um conjunto de casos de teste. Este estudo utiliza a idéia proposta em [9], onde foram utilizados caminhos de execução de testes para identificar comportamentos. É incluído um novo caso de teste e assim, caso esse revele um novo comportamento não identificado com o conjunto original, esse caso de teste é incluído no conjunto.

O programa é definido através de comportamentos em alto nível, utiliza a técnica *Category Partition*. Foi demonstrado um exemplo utilizando o programa *Triângulo* [45], o qual classifica o tipo do triângulo e calcula sua área a partir do valor de suas arestas. No exemplo apresentado, ao invés de utilizar casos de teste reais como este: (*a=2, b=3 e c=3*), são utilizadas regras como a seguinte: ($a \leq b + c, b = c$, *isósceles*). Sendo assim, têm-se as classes de equivalência de saída. Essa descrição de comportamentos em

alto nível foi feita para facilitar a leitura do algoritmo de AM, o qual aprende melhor as relações entre as entradas e saídas.

A metodologia utiliza o algoritmo *C4.5 (Decision Trees)* disponível no *Weka*. O nível de granularidade das classes de equivalência é decidido pelo testador. Por exemplo para o programa *Triângulo*, podem ser definidas quatro classes: eqüilátero, isósceles, irregular, não triângulo. Ou pode-se definir duas classes: triângulo ou não triângulo.

Através das regras são identificadas redundâncias nos casos de teste ou falta de casos de teste para uma certa característica. A ferramenta é iterativa, ou seja, é executada diversas vezes encontrando problemas através das regras e atualizando o conjunto de casos de teste, até que as regras não revelem mais problemas. Obrigatoriamente os casos de teste deverão ser executados sobre uma versão correta do programa para garantir que as saídas estejam corretas. Se um caso de teste revela um defeito, esse deve ser corrigido, o caso de teste executado novamente e, em seguida, o algoritmo *C4.5* deve ser executado para geração das regras.

A identificação dos problemas utilizando as regras é feita por meio de problemas pré-definidos. Também foram estabelecidas possíveis causas para cada um dos problemas pré-definidos. Sendo assim, o programa quando é executado, identifica um problema e uma ou mais possíveis causas para esse problema. De acordo com o problema identificado (casos de teste ausentes, casos de teste redundantes, etc), são tomadas ações corretivas: incluir casos de teste para uma determinada característica, eliminar redundâncias de casos de teste, etc. Para incluir novos casos de teste, são utilizadas heurísticas com base no resultado da árvore *C4.5*.

No estudo de caso realizado, foi utilizado o programa *PackHexChar* escrito em Java, utilizado no *GhostScript*. Foram identificadas 11 categorias, 23 possibilidades de escolha e 221 casos de teste foram utilizados no experimento. O estudo de caso mostrou que iniciando com um conjunto de casos de teste incompleto é possível encontrar diversos problemas, possibilitando o incremento e melhoria do conjunto de casos de teste. A taxonomia dos possíveis problemas e das possíveis causas se mostrou eficaz para este estudo de caso, mas precisa de mais experimentos para ser validada.

4.3.5 Seleção de Casos de Teste com Multi-Objetivos Utilizando Pareto [96]

O estudo realizado por Yoo e Harman [96] introduz o conceito de eficiência de Pareto para seleção de casos de teste. Essa abordagem toma múltiplos objetivos como cobertura de código, histórico de detecção de defeitos e custo de execução para selecionar um conjunto de casos de teste para o teste de regressão. São demonstrados os potenciais benefícios da abordagem de seleção proposta e ilustrados por meio de estudos empíricos com dois e três formulações de objetivos.

Nos estudos foram utilizados dois Algoritmos Genéticos: NSGA-II e sua variação vNSGA-II, implementada pelos autores. A saída do algoritmo NSGA-II não é uma solução única, mas o estado final da fronteira de Pareto que o algoritmo construiu. A otimização de Pareto é utilizada no processo de seleção dos indivíduos e o algoritmo tenta obter uma fronteira de Pareto maior através da seleção de indivíduos que estão distantes uns dos outros. A maior modificação implementada para a variação vNSGA-II é a utilização de um grupo de sub-populações que são separadas para obter fronteiras de Pareto maiores. Além dos Algoritmos Genéticos, também foram implementados dois algoritmos do tipo *Greedy*. Para a formulação com dois objetivos uma versão do algoritmo *Additional Greedy* foi implementada, já para a formulação com três objetivos, tais objetivos foram combinados em um único, através da abordagem clássica de soma de pesos.

Para a formulação com dois objetivos foram utilizados como critérios a cobertura de comandos e o custo computacional dos casos de teste. Já para a formulação com três objetivos, além dos critérios utilizados para a formulação com dois objetivos, foi utilizado o histórico de detecção de defeitos.

Os resultados empíricos obtidos revelam que os algoritmos do tipo *Greedy* nem sempre podem ser considerados Pareto eficientes no paradigma multi-objetivo, visto que tais algoritmos funcionam melhor para formulações com objetivo simples. Sendo assim, tal resultado motiva o estudo de novas técnicas, tais como técnicas de otimização multi-objetivo baseadas em meta-heurística. Os dois algoritmos genéticos produziram uma

fronteira de Pareto maior em relação aos algoritmos *Greedy*, o que já era esperado, visto que tais algoritmos foram desenvolvidos com esta finalidade.

4.3.6 Aprendizado Ativo para Classificação Automática de Comportamentos de Software [9]

O estudo realizado por Bowring et. al [9] propõe dois focos. O primeiro consiste em um método que utiliza aprendizado ativo para classificar automaticamente os comportamentos de um programa através dos dados de execução do programa. No aprendizado ativo, o classificador é treinado incrementalmente com uma série de elementos de dados rotulados. Já o segundo foco explora a tese de que certas funcionalidades do comportamento do programa são processos estocásticos que exibem a propriedade de *Markov*, e que os modelos de *Markov* resultantes de execuções individuais do programa podem ser automaticamente agrupados em preditores de comportamentos de programas.

O artigo apresenta uma técnica que modela execuções de programas em modelos de *Markov* e um método de agrupamento para modelos de *Markov* que agrega múltiplas execuções de programas em classificadores efetivos de comportamentos. Foi apresentada uma aplicação da técnica proposta que refina os classificadores utilizando *bootstrapping*, e tal aplicação foi ilustrada em um cenário no qual se pode reduzir os custos e ajudar a quantificar os riscos do teste de software, do desenvolvimento e da implantação. Por fim, foram conduzidos três estudos empíricos para avaliar a técnica que utiliza aprendizado ativo.

A vantagem do aprendizado ativo no contexto da engenharia de software é a habilidade de tornar mais eficiente o uso dos recursos limitados. Que são disponibilizados para analisar e rotular dados de execução de programas. Foi verificado empiricamente que o aprendizado ativo pode produzir classificadores de qualidade se comparado com o aprendizado por lote e ainda, com menos esforço para rotulagem dos dados. Além disso, foi verificado que os ítens de dados selecionados durante a aprendizagem ativa podem ser utilizados de forma eficiente para aumentar o escopo de planos de teste. Esses são resultados importantes, visto que uma análise humana de saídas de um programa é ine-

rentemente cara e muitas técnicas de AM necessitam de dados rotulados para obterem sucesso. Uma segunda contribuição do estudo é uma investigação das funcionalidades derivadas dos modelos de *Markov* que podem caracterizar adequadamente um conjunto de comportamentos, tais como as induzidas por um plano de teste.

4.3.7 Um Algoritmo Genético para Redução de Conjunto de Casos de Teste

O estudo realizado por Ma et. al [59] apresenta um Algoritmo Genético com base em um modelo matemático para realizar a redução de um conjunto de casos de teste para o teste de regressão. O enfoque do artigo é a redução de custo de execução. De acordo com o artigo, os trabalhos relacionados são voltados para cobertura estrutural, efetividade na descoberta de defeitos e análise de riscos ou até combinações desses aspectos. Contudo, a questão de custo de execução não é focada por causa de sua complexidade. Para tanto, o trabalho proposto utiliza um Algoritmo Genético com embasamento de um modelo matemático.

Diferente dos demais algoritmos descritos na literatura, o algoritmo proposto por Ma et. al é baseado em novos critérios. Tais critérios consistem em uma combinação de critérios baseados em cobertura de blocos e critérios de custo de execução de testes. Assim, é possível tomar decisões sobre a redução de um conjunto de casos de teste. O modelo matemático proposto é convertido em um problema de programação linear. Além disso, existe uma função que é utilizada para calcular a cobertura de conjuntos de casos de teste que pode ser modificada, assumindo diferentes critérios de cobertura, o que deixa o algoritmo flexível.

O artigo também apresenta os estudos empíricos realizados para avaliar o algoritmo proposto. Tais estudos demonstram que o algoritmo pode reduzir significativamente o tamanho e custo de um conjunto de casos de teste para o teste de regressão. A inclusão de critérios relacionados à custo para reduzir um conjunto de casos de teste mostrou-se eficiente para redução de custo e essa é uma característica muito importante que deve ser levada em consideração em qualquer redução de conjuntos de casos de teste para teste de

regressão.

Os autores explicam que, embora os resultados obtidos sejam bons, é necessário continuar o trabalho avaliando a capacidade de detecção de defeitos para o conjunto reduzido. Tais estudos validariam os algoritmos e forneceriam orientações ainda mais precisas para redução de um conjunto de casos de teste de regressão.

4.4 Considerações Finais

Este capítulo apresentou os principais conceitos e técnicas relacionados ao teste de regressão. Também foram expostos os estudos recentes que utilizam técnicas de AM realizados na área.

Conforme apresentado, diversos trabalhos utilizando AM, no contexto do teste de regressão, foram propostos e são considerados relacionados ao estudo realizado neste trabalho. Tais estudos têm objetivos distintos e utilizam informações e técnicas diversas para cumprir tais objetivos. Três dos trabalhos citados são relacionados à redução de um conjunto de casos de teste para o teste de regressão, destes, um utiliza Redes *Info-Fuzzy*, outro utiliza Redes Neurais e o último utiliza Algoritmos Genéticos. Um dos trabalhos é relacionado à priorização e outro à seleção de um conjunto de testes para o teste de regressão e estes, por sua vez, utilizam Algoritmos Genéticos e *Greedy*. Outros dois trabalhos relacionados inserem-se no contexto da classificação de dados. O primeiro visa a melhoria de um conjunto de testes para o teste de regressão utilizando árvores de decisão e o segundo classifica comportamentos de software utilizando aprendizado ativo e refinamento de classificadores utilizando *bootstrapping*.

A abordagem descrita no próximo capítulo, sendo proposta neste trabalho, utiliza técnicas de agrupamento para identificar classes de equivalência utilizando os dados de execução de casos de teste relacionados às técnicas estrutural e baseada em defeitos.

A abordagem proposta a seguir utiliza os dados oriundos das três técnicas de teste e, através de um classificador, gera regras. Essa parte da abordagem é similar à geração de regras utilizando árvore de decisão (*C4.5*) e também aos trabalhos que utilizam Redes Neurais e *Info-Fuzzy* para redução de casos de teste. Nota-se que nenhum dos trabalhos

relacionados apóia o teste de regressão utilizando as três técnicas de teste como base. Outro fator importante é que as classes de equivalência e regras geradas possibilitam seleção, priorização e redução de um conjunto de casos de teste, não restringindo-se somente à uma abordagem de teste de regressão.

CAPÍTULO 5

ABORDAGEM E FERRAMENTA

Neste capítulo é apresentada a abordagem genérica proposta para geração de regras que relacionam os dados das diferentes técnicas de teste. Em seguida é apresentada a ferramenta RITA (***R****elating* ***I****nformation from* ***T****esting* ***A****ctivity*) que implementa a abordagem proposta, demonstrando sua estrutura e ambiente. Por fim, são apresentadas as formas de utilização das regras geradas para apoiar o teste de regressão.

5.1 Abordagem Proposta

Esta abordagem tem como entrada um conjunto de casos de teste T e informações dele derivadas para um programa P, as quais são obtidas a partir de critérios estruturais e baseados em defeitos. Essas informações da atividade de teste são utilizadas para identificação de classes de equivalência. Em seguida, a abordagem propõe a utilização de um classificador para geração de regras que estabelecem a relação entre as diferentes técnicas de teste: estrutural e baseada em defeitos com a técnica funcional, para a qual é considerado o critério particionamento em classes de equivalência. Na Figura 5.1 é apresentado um fluxograma representativo da abordagem proposta.

Conforme abordado no Capítulo 3, o critério particionamento em classes de equivalência, da técnica funcional, consiste em decompor o conjunto de entradas em uma quantidade finita de classes de equivalência que permitem a escolha de um valor de teste representativo para cada classe. O teste que resulta do valor representativo para uma classe é chamado de equivalente para os outros valores na mesma classe. Tendo em vista a ampla utilização desse critério de teste, foi proposta uma abordagem que utiliza AM para identificar automaticamente as classes de equivalência de um determinado programa P, a partir das diferentes informações produzidas pela atividade de teste. Para cada caso de teste $t \in T$ executado, são armazenados, em forma de tabela, os dados listados a seguir:

- Entradas: Cada uma das entradas de t;
- Saídas: Cada uma das saídas da execução de t em P;
- Porcentagem de Cobertura de Critérios Estruturais: Porcentagem de cobertura de elementos requeridos pelos critérios estruturais cobertos pela execução de t em P;
- Número de Mutantes Mortos: Considerando a geração de mutantes para todos os operadores de mutação possíveis, é utilizado o total de mutantes mortos por t;
- Porcentagem de Cobertura de Classes de Defeitos: Para cada um das classes de defeitos (cada operador de mutação é considerado uma classe de defeito), calcula-se a porcentagem de cobertura a partir do total de mutantes gerados para cada classe e o total de mutantes mortos por t. São desconsiderados os operadores de mutação que não geraram mutantes e os operadores para os quais todos os casos de teste geram a mesma porcentagem de cobertura.

Aplicando os algoritmos de agrupamento apresentados no Capítulo 2, sobre o conjunto de dados apresentado, é possível obter as classes de equivalência para o programa P. Sendo assim, têm-se também a associação de cada um dos casos de teste de T com sua respectiva classe de equivalência.

O agrupamento gerado relaciona os dados das três técnicas de teste. Além disso, ele provê um atributo nominal (classe de equivalência) que pode ser utilizado como atributo a ser predito por um algoritmo de classificação. Um algoritmo de classificação, conforme exposto no Capítulo 2, possibilita a geração de regras e essas regras podem ser muito úteis junto ao teste de regressão.

A derivação de regras que podem ser utilizadas na engenharia de software como meios de calibrar a seleção de casos de teste de regressão é um problema em aberto. Sendo assim, a abordagem proposta neste trabalho serve para apoiar o teste de regressão e a gerência de projetos na tomada de decisões sobre a atividade de teste e manutenção de software.

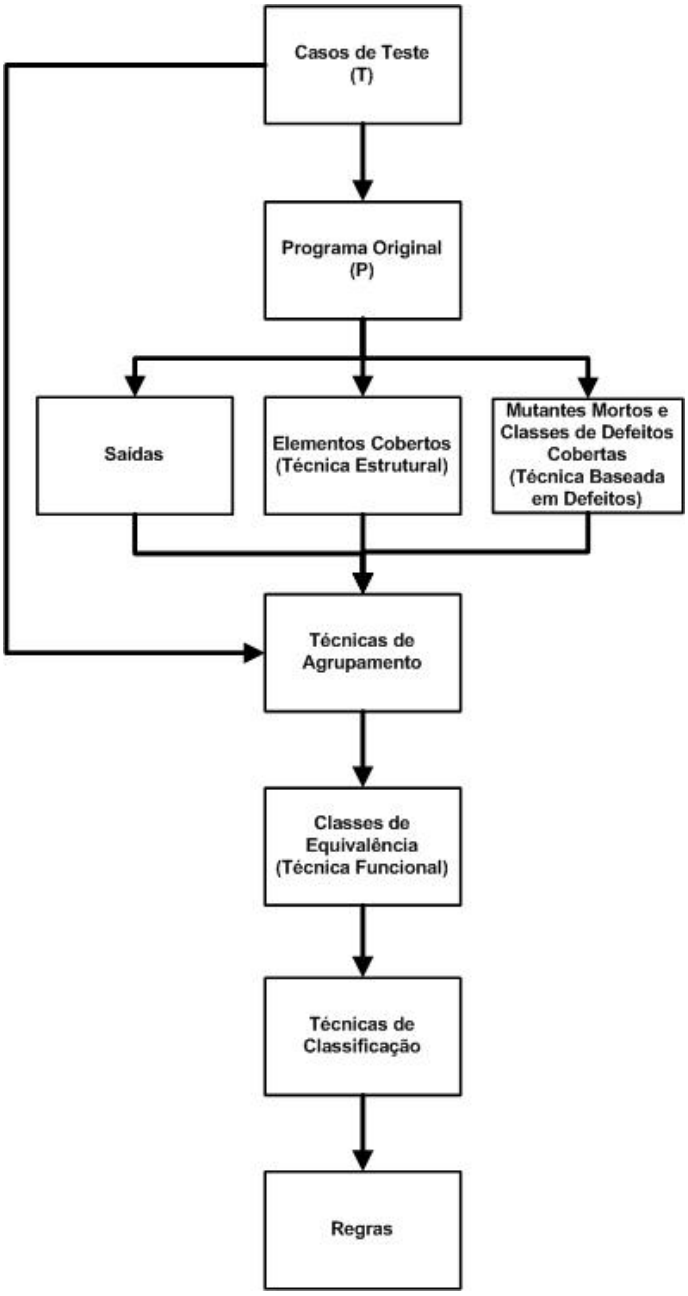


Figura 5.1: Fluxograma representativo da abordagem

5.2 Exemplo de Utilização

Esta seção apresenta um exemplo dos dados oriundos de cada uma das etapas do fluxograma apresentado na Figura 5.1, observando-se que tais dados são fictícios. Para tal exemplo, considera-se um programa P que possui duas entradas numéricas. Tal programa retorna um valor numérico correspondente à multiplicação dos valores das duas entradas. A seguir, um exemplo de dados, correspondente ao programa em questão, é apresentado para cada uma das etapas do fluxograma. As tabelas apresentadas são cumulativas, sendo que a última tabela contém todos os dados que são utilizados para geração das regras.

5.2.1 Casos de Teste (T)

O conjunto T representa dados de teste para testar o programa P. Um exemplo de conjunto T pode ser observado na Tabela 5.1.

Tabela 5.1: Exemplo de dados de teste

Entrada 1	Entrada 2
0	13
-5	16
2	7
-2	-3
7	-1
4	4
9	0

5.2.2 Saídas

As saídas representam a execução do programa P para os dados de T os quais são apresentados na Tabela 5.2.

5.2.3 Elementos Cobertos (Técnica Estrutural)

Os elementos cobertos referem-se à quaisquer critérios da técnica estrutural. Neste exemplo são utilizados os critérios PU e PDU, onde são utilizadas as porcentagens de cobertura

Tabela 5.2: Exemplo saídas

Entrada 1	Entrada 2	Saída
0	13	0
-5	16	-80
2	7	14
-2	-3	6
7	-1	-7
4	4	16
9	0	0

para cada dado de teste e, estas, por sua vez, são apresentas na Tabela 5.3.

Tabela 5.3: Exemplo de elementos cobertos (técnica estrutural)

Entrada 1	Entrada 2	Saída	% PU	% PDU
0	13	0	42	23
-5	16	-80	64	60
2	7	14	84	75
-2	-3	6	84	75
7	-1	-7	64	60
4	4	16	84	75
9	0	0	42	23

5.2.4 Mutantes Mortos e Classes de Defeitos Cobertas (Técnica Baseada em Defeitos)

As classes de defeitos cobertas referem-se à quaisquer operadores de mutação. Neste exemplo são utilizados os operadores Cccr e OAAA (descritos no Apêndice A), onde são utilizadas as porcentagens de cobertura de mutantes mortos para cada dado de teste e, estas, por sua vez, são apresentas na Tabela 5.4, juntamente com o número total de mutantes mortos por cada dado de teste.

Tabela 5.4: Exemplo de mutantes mortos e classes de defeitos cobertas (técnica baseada em defeitos)

Entrada 1	Entrada 2	Saída	% PU	% PDU	% Cccr	% OAAA	Mut. Mortos
0	13	0	42	23	67	23	2340
-5	16	-80	64	60	88	60	2923
2	7	14	84	75	88	60	2923
-2	-3	6	84	75	20	60	2923
7	-1	-7	64	60	88	60	2923
4	4	16	84	75	18	60	2923
9	0	0	42	23	67	23	2340

5.2.5 Técnicas de Agrupamento e Classes de Equivalência (Técnica Funcional)

Utilizando como base os exemplos de dados da Tabela 5.4, a abordagem propõe a aplicação de uma técnica de agrupamento para identificar as classes de equivalência do programa P. Diversas técnicas de agrupamento podem ser usadas, algumas possibilidades são descritas no Capítulo 2. O resultado da aplicação da técnica de agrupamento é a associação de cada dado de teste a uma classe de equivalência, conforme apresentado na Tabela 5.5.

Tabela 5.5: Exemplo de classes de equivalência (técnica funcional)

Entrada 1	Entrada 2	Saída	% PU	% PDU	% Cccr	% OAAA	Mut. Mortos	Classe Equivalência
0	13	0	42	23	67	23	2340	cluster1
-5	16	-80	64	60	88	60	2923	cluster2
2	7	14	84	75	88	60	2923	cluster3
-2	-3	6	84	75	20	60	2923	cluster3
7	-1	-7	64	60	88	60	2923	cluster2
4	4	16	84	75	18	60	2923	cluster3
9	0	0	42	23	67	23	2340	cluster1

5.2.6 Técnicas de Classificação e Regras

Por fim, a fluxograma apresenta a aplicação de uma técnica de classificação (algumas técnicas de classificação são apresentadas no Capítulo 2) para identificação de regras, as

quais podem ser aplicadas para apoio ao teste de regressão. Tais algoritmos geralmente fornecem regras de fácil entendimento no formato *Se-Então*. Como exemplos de possíveis regras têm-se:

- Se % PU = 60, Então cluster2;
- Se % OAAA = 23, Então cluster1;
- Se Saída > 0, Então cluster3;

5.2.7 Utilizando as Regras para Redução, Seleção e Priorização

As regras geradas pelo classificador, as quais relacionam os dados obtidos do teste estrutural, baseado em defeitos e funcional, podem ser utilizadas de diferentes maneiras. A seguir, cada uma dessas formas de utilização são descritas.

5.2.7.1 Redução

A redução do conjunto de casos de teste para o teste de regressão pode ser melhorada através das regras geradas. Tendo em vista que, utilizando a abordagem convencional, o testador escolheria somente um caso de teste de cada classe para o teste de regressão, utilizando as regras o testador poderá identificar características específicas de cada classe de equivalência, verificando que algumas podem ser melhores que outras, dependendo do aspecto que ele deseja avaliar. Por exemplo, um caso de teste de uma determinada classe de equivalência pode ter porcentagem de cobertura de um critério estrutural superior a todas as demais classes de equivalência e ainda cobrir uma determinada classe de defeitos, enquanto as demais classes de equivalência não cobrem. Informações desse tipo são úteis para ajudar o testador a indentificar os melhores casos de teste para o teste de regressão.

5.2.7.2 Seleção

Conforme exposto no Capítulo 2, várias técnicas de seleção foram propostas. Dentre essas algumas são baseadas somente na especificação do programa e outras utilizam informações

do código-fonte. Elas possuem diferentes objetivos: técnicas baseadas em cobertura, técnicas de minimização e técnicas seguras. A abordagem proposta nesse trabalho também utiliza informações de código-fonte e também pode ser utilizada para cumprir os objetivos citados. O testador deverá identificar o aspecto a ser enfatizado na seleção e, com base nesse aspecto, as regras são utilizadas para indicar quais os melhores casos de teste.

O testador pode identificar as classes de equivalência mais importantes, ou seja, classes que cobrem trechos críticos do programa ou que possuem uma maior importância em relação aos critérios de teste, selecionando assim mais casos de teste dessa classe, outras possibilidades de aspectos que podem ser avaliados pelo testador são: maximização de cobertura de um determinado critério estrutural, cobertura de classes de defeitos, entre outras.

5.2.7.3 Priorização

Assim como na redução e na seleção de um conjunto de casos de teste, a priorização também depende do aspecto que o testador deseja enfatizar. As regras geradas permitem ao testador escolher um determinado aspecto. Por exemplo, o testador poderá definir que deseja enfatizar a cobertura das classes de defeitos do operador de mutação em variáveis. Sendo assim, analisando as regras, é possível definir quais casos de teste devem ser priorizados para maximizar a cobertura dessas classes de defeitos.

5.3 Implementação da Abordagem - RITA

Com o intuito de validar a abordagem proposta, foi desenvolvida uma ferramenta chamada RITA. A ferramenta RITA necessita de diversos dados que devem ser fornecidos pelo testador: um programa P, a função de P que será testada, o tipo de entrada que P suporta, o diretório onde encontra-se um conjunto de casos de teste T para P e a quantidade de casos de teste contidos no diretório informado. Na Figura 5.2 é apresentada a interface gráfica da RITA.

A RITA foi desenvolvida utilizando a linguagem de programação Java e opera no

Figura 5.2: Interface gráfica da RITA

sistema operacional Linux. Ela integra as ferramentas descritas na Seção 3.4: *Poke-Tool* (***Po****tential-Uses* ***Cri****teria* ***Tool*** *for program testing*) versão **script** 1.1 e *Proteum*(***Pro****gram* ***Te****sting* ***U****sing* ***M****utants*) versão **script** 1.4.1. A RITA possui as mesmas restrições que as ferramentas citadas, por exemplo, não são aceitos programas que possuem mais de um nó de saída. Esta restrição é herdada da *Poke-Tool* e maiores detalhes sobre funcionamento, restrições e configurações da *Poke-Tool* e *Proteum* podem ser vistos em [14, 21], respectivamente.

A ferramenta desenvolvida utiliza um diretório padrão /rita/. Ao executar a RITA, dentro desse diretório é criado um novo diretório com o nome do programa em teste, em seguida o programa é copiado para o diretório criado. Dois novos diretórios são criados dentro do diretório do programa: /pok/ e /pro/ onde são armazenados os dados de execução da *Poke-Tool* e da *Proteum*, respectivamente.

Ao finalizar a criação de diretórios, inicia-se a execução da *Poke-Tool*. Tanto a *Poke-Tool* quanto a *Proteum* são acionadas através de *shell scripts*. Na Figura 5.3 é apresentada a arquitetura da ferramenta RITA. A execução da *Poke-Tool*, conforme ilustrada na arquitetura, se dá em 5 passos:

1. Esta etapa consiste em executar o comando *poketool* para o programa. Tal comando tem como finalidade obter os elementos requeridos para cada um dos critérios estruturais, assim como instrumentar o programa;

2. Utilizando o compilador C disponível no ambiente, o programa instrumentado na etapa anterior (*testeprog.c*) é compilado;

3. Para ler e executar os casos de teste do conjunto T, a RITA utiliza o tipo de entrada (parâmetro ou teclado), visto que a nomenclatura dos casos de teste é diferente para cada tipo. Também é utilizada a quantidade de casos de teste a ser executada e o diretório onde encontram-se os casos de teste. Com esses dados, a ferramenta lê cada um dos casos de teste e os executa na *Poke-Tool* para o programa instrumentado através do comando *pokeexec*. Essa execução gera o caminho percorrido pelo caso de teste;

4. A avaliação de cobertura de cada caso de teste é realizada através do comando *pokeaval*. É utilizado o caminho percorrido por cada caso de teste para avaliar a cobertura do mesmo em relação aos elementos requeridos de cada critério estrutural;

5. Para cada caso de teste executado, são armazenadas as porcentagens de cobertura para os seguintes critérios estruturais: Todos-Nós, Todos-Arcos, Todos-Potenciais-Usos (PU), Todos-Potenciais-Du-Caminhos (PDU) e Todos-Potenciais-Usos/DU (PUDU). Também são armazenados os dados de entrada para o programa e suas respectivas saídas geradas.

Após o término da execução da *Poke-Tool*, a RITA aciona a *Proteum*, essa execução segue uma seqüência de 7 passos:

1. Inicialmente o programa P é compilado utilizando o compilador C disponível no ambiente;

2. É gerada um nova seção de teste para o programa utilizando o comando *test-new*;

3. São gerados os mutantes para todos os 71 operadores de mutação disponíveis utilizando o comando *muta-gen*;

4. Os mesmos casos de teste executados na *Poke-Tool* são importados para a *Proteum* através do comando *tcase -poke*;

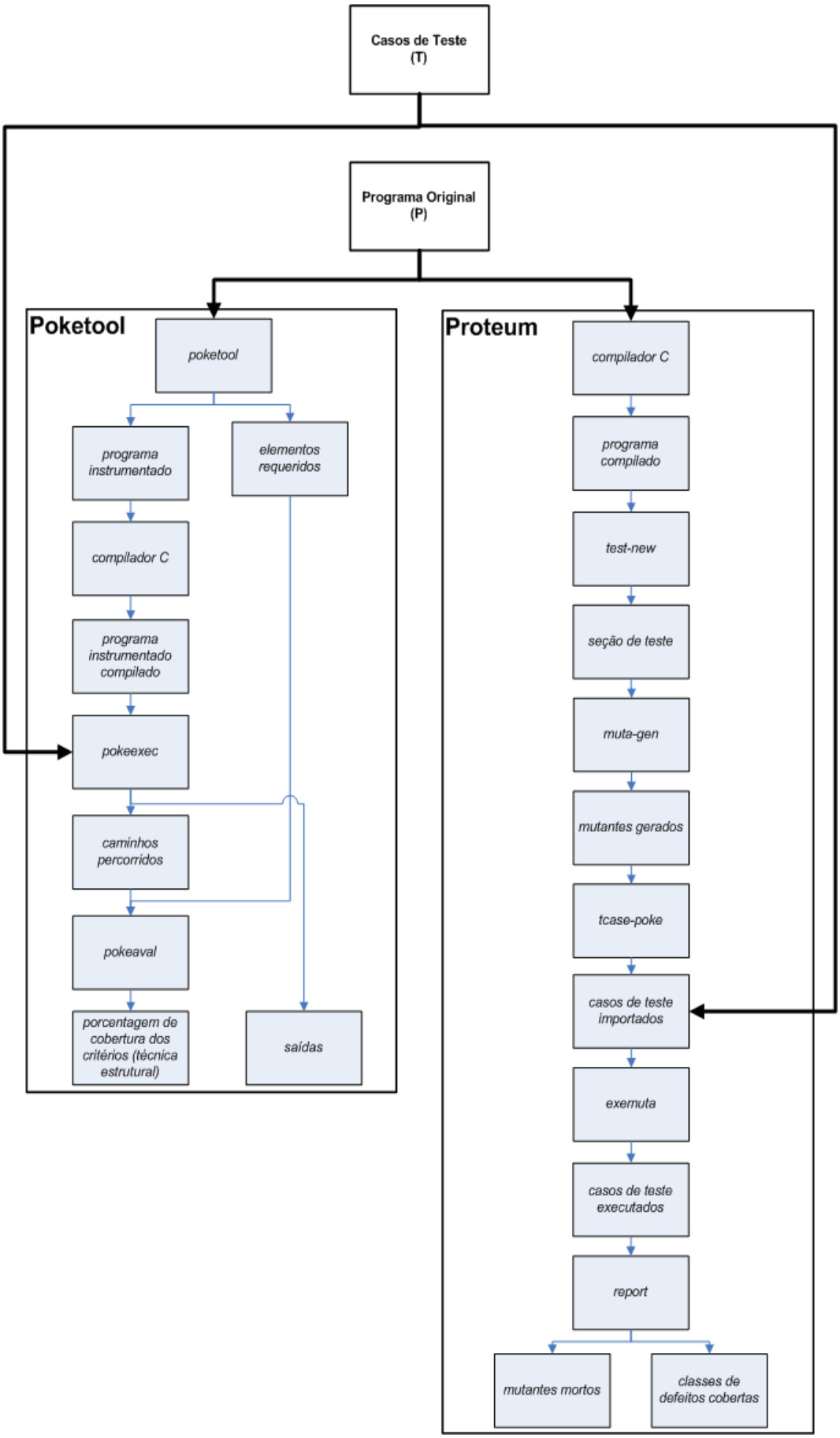


Figura 5.3: Arquitetura da Ferramenta RITA

5. Todos os casos de teste importados são desabilitados, para que, em seguida, um a um seja habilitado para execução, uma vez que cada um deve ser avaliado individualmente;

6. Através do comando *exemuta* cada um dos casos de teste é executado sobre todos os mutantes gerados;

7. Para cada caso de teste executado, é gerado um relatório através do comando *report*. Este relatório contém o número de mutantes mortos e quais os mutantes que foram mortos, possibilitando a identificação de quais operadores de mutação que foram cobertos e, conseqüentemente, a porcentagem de cobertura.

Após a execução das duas ferramentas, a RITA agrupa todas as informações obtidas em um arquivo. Cada um dos casos de teste executados na *Poke-Tool* e na *Proteum* representa uma linha do arquivo e cada uma das informações obtidas da execução representa uma coluna. Sendo assim, é obtida uma tabela contendo todas as informações de execução dos casos de teste. Este arquivo é gerado no formato *.arff* para que possa ser utilizado na ferramenta *Weka*, possibilitando a próxima etapa da abordagem referente à aplicação dos algoritmos de agrupamento.

As duas últimas fases da abordagem referem-se à utilização da ferramenta *Weka* e são executadas manualmente. A identificação de classes de equivalência utiliza uma das três técnicas de agrupamento descritas no Capítulo 2: *K-Means*, *COBWEB* ou *EM* e toma como base o arquivo gerado pela RITA. Foram escolhidas essas três técnicas de agrupamento pois elas utilizam estratégias diferentes. O algoritmo *K-Means* utiliza erro quadrático, já o *EM* utiliza resolução de misturas e o *COBWEB* utiliza modelo conceitual incremental. Através da execução dos algoritmos utilizando o *Weka*, é gerado um novo arquivo .arff onde cada caso de teste está associado à sua respectiva classe de equivalência (*cluster*). Em seguida, esse novo arquivo gerado é utilizado como entrada para o algoritmo de classificação *J48*, o qual gera as regras. Esse algoritmo necessita de um atributo nominal para ser utilizado como atributo de classificação ou alvo e, nesse caso, é utilizado o atributo *cluster*. As regras inferidas pelo classificador são geradas em função desse

atributo de classificação.

5.4 Considerações Finais

A abordagem proposta, diferentemente dos diversos trabalhos utilizando AM para apoio ao teste de regressão, possibilita a utilização das regras para redução, seleção e priorização de casos de teste. Assim, a abordagem não é restrita a uma aplicação, tornando-se ampla no contexto do teste de regressão. Além disso, a abordagem proposta neste trabalho utiliza técnicas de agrupamento para identificar classes de equivalência, utilizando os dados de execução de casos de teste relacionados às técnicas estrutural e baseada em defeitos.

São utilizados os dados oriundos das três técnicas de teste e, através de um classificador, a abordagem permite a geração de regras. Essa parte da abordagem é similar à geração de regras utilizando árvore de decisão (*C4.5*) e também aos trabalhos que utilizam Redes Neurais e *Info-Fuzzy* para redução de casos de teste. Contudo, nota-se que nenhum dos trabalhos relacionados apóia o teste de regressão utilizando as três técnicas de teste como base.

As técnicas de teste utilizadas na abordagem são consideradas complementares por revelarem diferentes tipos de defeitos. A ferramenta RITA auxilia a aplicação dos critérios dessas técnicas, possibilitando um ambiente integrado através do acionamento das ferramentas *Poke-Tool* e *Proteum*. A percepção e utilização das relações entre as informações geradas pelas duas ferramentas era inviável por um usuário testador, contudo, agora tornou-se possível através da ferramenta desenvolvida. Assim, tais relações podem ser utilizadas para apoiar o teste de regressão.

Embora a ferramenta RITA demonstre diversas contribuições, ela também possui algumas limitações. Dentre elas, pode-se citar o tempo gasto para sua execução que é muito elevado. Devido ao fato da RITA acionar a ferramenta *Proteum* para avaliar cada caso de teste individualmente, o tempo de execução aumenta significativamente, podendo chegar a uma hora quando executada para um conjunto de 400 casos de teste. Outra dificuldade apresentada é referente à sua configuração, por tratar-se de um ambiente que integra duas ferramentas, ela necessita da instalação e configuração correta das ferramentas *Poke-Tool*

e *Proteum* para poder funcionar corretamente.

Conforme exposto, a ferramenta e abordagem proposta apresentam diversas contribuições. Para tanto, necessita-se a avaliação e validação da abordagem proposta. O próximo capítulo apresenta tal avaliação através da condução de experimentos utilizando a ferramenta RITA. Também é apresentada a aplicação das regras geradas nos experimentos conduzidos para apoio ao teste de regressão.

CAPÍTULO 6

AVALIAÇÃO DA ABORDAGEM

A abordagem foi aplicada a quatro programas escritos na linguagem C utilizando a RITA com o objetivo de validar as idéias apresentadas. As seções seguintes apresentam a metodologia utilizada para realização dos experimentos, identificação de classes de equivalência, a geração de regras e a aplicação das regras geradas.

6.1 Metodologia

Essa seção apresenta a metodologia utilizada para realização dos experimentos. Essa metodologia é composta por: objetivo dos experimentos, descrição dos programas utilizados, geração dos casos de teste, combinações de atributos, parâmetros de configuração das técnicas de agrupamento e passos realizados. Essa metodologia descrita é utilizada para todos os experimentos que foram conduzidos.

6.1.1 Objetivo dos Experimentos

O objetivo dos experimentos é dividido em três partes: 1) identificar classes de equivalência utilizando os dados de execução de testes e avaliar as classes geradas em relação às classes definidas manualmente; 2) utilizar a classificação gerada e os dados de execução dos testes para inferir regras; 3) utilizar as regras geradas para apoio ao teste de regressão através de priorização, seleção e redução de conjuntos de casos de teste. Os novos conjuntos de casos de teste gerados são avaliados através de comparação com as abordagens *retest-all* e padrão (seleção de um caso de teste para cada classe de equivalência).

Para o objetivo 1, foram definidos limiares de resultados, os quais referem-se à porcentagem de acerto dos experimentos na identificação de classes de equivalência:

- Bons: aqueles experimentos que obtiveram acerto superior à 90%;

- Satisfatórios: aqueles experimentos que obtiveram acerto superior à 80% e inferior ou igual à 90%;
- Ruins: aqueles experimentos que obtiveram acerto inferior ou igual à 80%.

6.1.2 Descrição dos Programas

Conforme citado anteriormente, foram utilizados quatro programas escritos na linguagem *C* para realização dos experimentos: *Triângulo* [45], *Bublle Sort* [68, 69], *GetCmd* [47] e *FourBalls* [68, 69]. Nesta seção são apresentados os programas juntamente com uma breve descrição das particularidades de cada um.

- *Triângulo*: O programa *Triângulo* classifica o tipo do triângulo e calcula sua área a partir do valor de suas arestas.
- *Bubble Sort*: O programa *Bubble Sort* ordena cinco valores numéricos.
- *FourBalls*: O programa *FourBalls* recebe como entrada 4 valores para as bolas e um limiar. Dependendo do valor do limiar, é realizado um cálculo diferente como peso das bolas.
- *GetCmd*: O programa *GetCmd* retorna dois caracteres que correspondem ao comando associado aos caracteres fornecidos como entradas.

6.1.3 Geração de Casos de Teste

Para cada programa foram gerados 500 casos de teste aleatoriamente, exceto para o programa *GetCmd* que foram gerados 400. Cada caso de teste foi gerado em um arquivo individual no formato suportado pelas ferramentas *Poke-Tool* e *Proteum*. Vale ressaltar que o programa *GetCmd* possui entrada via parâmetro, diferente dos demais programas cujas entradas são via teclado. O formato dos casos de teste para este programa também são diferenciados.

Existem algumas particularidades que foram identificadas para os programas *FourBalls* e *GetCmd*. A seguir, tais particularidades são comentadas.

Partindo do princípio que atributos que possuem o mesmo valor para todas as instâncias não agregam valor à identificação de classes de equivalência, todos os atributos para os quais esse fato ocorre são removidos. Esse fato é bastante comum para a porcentagem de cobertura das classes de defeitos, onde operadores que não geram mutantes ficam com valor 0 para todos os casos de teste. Também, é freqüente a ocorrência de operadores onde todos os seus mutantes gerados são mortos por todos os casos de teste, gerando cobertura igual a 100 para todos os casos de teste.

O programa *FourBalls* é um programa muito simples e todos os casos de teste que foram gerados aleatoriamente para este programa geraram a mesma cobertura para os critérios estruturais. Assim, as porcentagens de coberturas dos critérios estruturais não foram utilizadas nos experimentos (exceto porcentagem de cobertura do critério Todos-Nós que foi utilizada). Com isto, as combinações de atributos de entrada para identificação das classes de equivalência foram reduzidas.

O mesmo ocorre para o programa *GetCmd* que gera a mesma porcentagem de cobertura dos critérios estruturais para todos os casos de teste. Além disso, somente o operador de mutação *Cccr* gerou porcentagens de cobertura diferentes para os casos de teste, sendo que os demais foram descartados. Esses fatos fazem com que as combinações de atributos de entrada para identificação das classes de equivalência sejam ainda mais reduzidas para este programa.

Para cada um desses programas foram definidas classes de equivalência e cada caso de teste foi pré-classificado manualmente em relação às classes de equivalência. As tabelas 6.1 à 6.4 apresentam as classes de equivalência geradas manualmente e o número de casos de teste em cada classe para cada um dos programas.

6.1.4 Combinações de Atributos e Parâmetros de Configuração das Técnicas de Agrupamento

Em virtude da grande quantidade de experimentos realizados, foi estabelecida uma forma de referência numérica aos experimentos. Inicialmente cada um dos programas foi numerado: 1) *Triângulo*, 2) *Bubble Sort*, 3) *FourBalls* e 4) *GetCmd*. Em seguida, cada uma

Tabela 6.1: Classes de equivalência manuais para o programa *Triângulo*

Classe Equivalência	Descrição	# Casos de Teste
0	Não forma triângulo	91
1	Triângulo equilátero	86
2	Triângulo isósceles	47
3	Triângulo Retângulo	84
4	Triângulo Agudo	105
5	Triângulo Obtuso	87
Total		500

Tabela 6.2: Classes de equivalência manuais para o programa *Bubble Sort*

Classes	Descrição	# Casos de Teste
0	Valores ordenados	130
1	Valores desordenados	112
2	Valores ordenados inversamente	117
3	Todos os valores iguais	141
Total		500

Tabela 6.3: Classes de equivalência manuais para o programa *FourBalls*

Classes	Descrição	# Casos de Teste
0	Limiar igual a -100	137
1	Limiar igual a 0	116
2	Limiar igual a 100	122
3	Limiar diferente dos demais valores	125
Total		500

das técnicas de agrupamento foi numerada: 1) *K-Means*, 2) *COBWEB* e 3) *EM*.

Também foram numeradas as combinações de atributos utilizadas nos experimentos, conforme segue:

1. Entradas e saídas;
2. Porcentagens de cobertura dos critérios estruturais;
3. Número de mutantes mortos e porcentagem de cobertura das classes de defeitos;
4. Entradas, saídas e porcentagens de cobertura dos critérios estruturais;
5. Entradas, saídas, número de mutantes mortos e porcentagem de cobertura das clas-

Tabela 6.4: Classes de equivalência manuais para o programa *GetCmd*

Classes	Descrição	# Casos de Teste
0	parâmetro de entrada igual a 99	31
1	parâmetro de entrada igual a 98	33
2	parâmetro de entrada igual a 97	30
3	parâmetro de entrada igual a 96	25
4	parâmetro de entrada igual a 95	28
5	parâmetro de entrada igual a 94	24
6	parâmetro de entrada igual a 93	24
7	parâmetro de entrada igual a 92	25
8	parâmetro de entrada igual a 91	29
9	parâmetro de entrada igual a 90	29
10	parâmetro de entrada igual a 89	19
11	parâmetro de entrada igual a 88	23
12	parâmetro de entrada igual a 87	27
13	parâmetro de entrada igual a 86	27
14	parâmetro de entrada com valor diferente dos anteriores	26
Total		400

ses de defeitos;

6. Porcentagens de cobertura dos critérios estruturais, número de mutantes mortos e porcentagem de cobertura das classes de defeitos;

7. Todos os atributos;

Portanto a numeração de cada experimento consiste em utilizar o número do programa, mais o número da técnica, mais o número da combinação de atributos. Por exemplo: 2.3.1 refere-se ao experimento conduzido para o programa *Bubble Sort*, utilizando a técnica *EM* e como combinação de atributos, as entradas e saídas. É importante ressaltar que as combinações citadas diferem para cada programa, em virtude das particularidades em relação aos atributos conforme exposto anteriormente, sendo considerados somente os resultados bons e satisfatórios.

A Tabela 6.5 demonstra os parâmetros utilizados para as técnicas de agrupamento em cada um dos experimentos bons e satisfatórios. Já os 34 experimentos que geraram resultados ruins não foram apresentados na tabela.

Nota-se que a variação dos valores dos parâmetros de um experimento para outro é

Tabela 6.5: Parâmetros para as técnicas de agrupamento

Experimento	clusters	cutoff	acuity	MaxIterations	MinStdDev
1.1.2	6				
1.2.2		0,08	0,9		
1.3.2	6			100	1,00E-08
1.2.3		0,8	1		
1.2.4		0,05	0,9		
1.2.5		0,8	1		
1.3.5	-1			100	1,00E-05
1.2.6		1	1		
1.3.6	-1			100	1,00E-06
1.2.7		1	1		
1.3.7	-1			100	1,00E-06
2.1.3	4				
2.2.3		0,5	0,9		
2.2.6		0,6	1		
2.3.6	4			100	1,00E-05
3.2.5		0,4	1		
3.3.5	-1			100	1,00E-06
3.2.6		0,4	1		
3.3.6	-1			100	1,00E-06
3.2.7		0,4	1		
3.3.7	-1			100	1,00E-06
4.1.7	15				
4.2.7		3.0E-5	0,9		

mínima. Esses valores foram encontrados através de tentativa e erro para cada programa, visto que uma alteração mínima nos parâmetros pode alterar totalmente o resultado obtido. Foi utilizado o método de tentativa e erro até que um resultado bom ou satisfatório fosse atingido, quando possível.

6.1.5 Condução dos Experimentos

Para início dos experimentos, executa-se a ferramenta RITA para cada um dos programas, a qual, conforme exposto no Capítulo 5, executa o conjunto de casos de teste acionando as ferramentas *Poke-Tool* e *Proteum*. A RITA gera um arquivo no formato suportado pelo *Weka* e sobre este arquivo são realizados os experimentos. Tais experimentos consistem em utilizar as três técnicas de agrupamento: *K-Means*, *COBWEB* e *EM* sobre as combinações

de atributos.

As técnicas de agrupamento têm como objetivo identificar as classes de equivalência de cada programa. Cada uma das técnicas de agrupamento possui parâmetros, apresentados anteriormente, que devem ser configurados para cada experimento. O resultado obtido por essa fase é um novo arquivo no formato suportado pela ferramenta *Weka* onde cada uma das instâncias está classificada, de acordo com as classes de equivalência obtidas.

Esse novo arquivo, por sua vez, é submetido novamente à ferramenta *Weka*, desta vez é utilizada uma técnica de classificação (*J48*). Essa técnica gera um conjunto de regras, as quais são úteis para melhoria de um conjunto de casos de teste para o teste de regressão. Tais melhorias se dão através da utilização das regras para seleção, priorização e redução de um conjunto de casos de teste.

6.2 Identificação de Classes de Equivalência

Nesta seção são apresentados os resultados quanto à identificação de classes de equivalência para os experimentos realizados nos programas: *Triângulo*, *Bublle Sort*, *FourBalls* e *GetCmd*.

6.2.1 *Triângulo*

Os experimentos conduzidos que geraram resultados satisfatórios quanto à identificação de classes de equivalência foram: 1.1.2, 1.2.2, 1.3.2, 1.2.3, 1.2.4, 1.2.5, 1.3.5, 1.2.6, 1.3.6, 1.2.7 e 1.3.7. Destes, todos geraram a mesma classificação (Figura 6.1), exceto o experimento 1.3.7 utilizando a técnica *EM* e todos os atributos de entrada que gerou o melhor resultado dentre todos experimentos, sendo que somente um caso de teste não foi classificado corretamente, conforme apresentado na Figura 6.2.

Pôde-se analisar que mais da metade destes experimentos geraram resultados bons ou satisfatórios, ou seja, 11 dos 21 experimentos realizados. Desses 11 experimentos, 10 obtiveram resultado final idêntico. A Figura 6.1 apresenta a comparação entre as classes de equivalência obtidas manualmente e as obtidas pelas técnicas de agrupamento.

Visto que os melhores resultados referentes aos dez experimentos são idênticos, o gráfico apresentado corresponde à esses dez experimentos. Analisando o gráfico pode-se observar que somente as classes de equivalência 4 e 5 não foram classificadas corretamente. As técnicas de agrupamento classificaram estas classes como sendo a mesma.

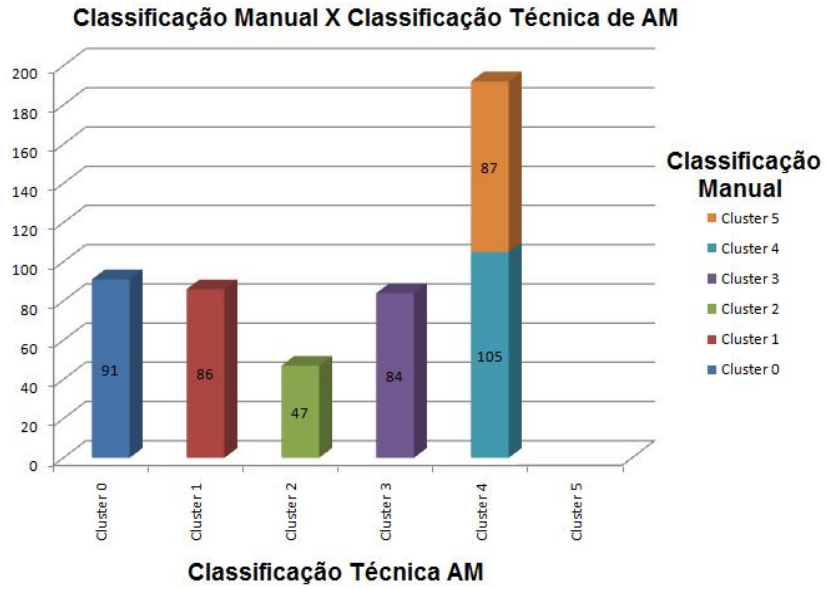


Figura 6.1: Gráfico comparativo de classes de equivalência para o programa *Triângulo* - Experimentos 1.1.2, 1.2.2, 1.3.2, 1.2.3, 1.2.4, 1.2.5, 1.3.5, 1.2.6, 1.3.6 e 1.2.7

6.2.2 *Bubble Sort*

Assim como para o programa *Triângulo*, para o *Bubble Sort* também foram realizados 21 experimentos. Desses 21, 4 geraram resultados satisfatórios ou bons, são eles: 2.1.3 (técnica: *K-means*, atributos: número de mutantes mortos e porcentagem de cobertura das classes de defeitos), 2.2.3 (técnica: *COBWEB*, atributos: número de mutantes mortos e porcentagem de cobertura das classes de defeitos), 2.2.6 (técnica: *COBWEB*, atributos: porcentagens de cobertura dos critérios estruturais, número de mutantes mortos e porcentagem de cobertura das classes de defeitos) e 2.3.6 (técnica: *EM*, atributos: porcentagens de cobertura dos critérios estruturais, número de mutantes mortos e porcentagem de cobertura das classes de defeitos).

As classificações obtidas pelas técnicas de agrupamento não correspondem exatamente às classes de equivalência definidas manualmente. Alguns dos casos de teste referentes à

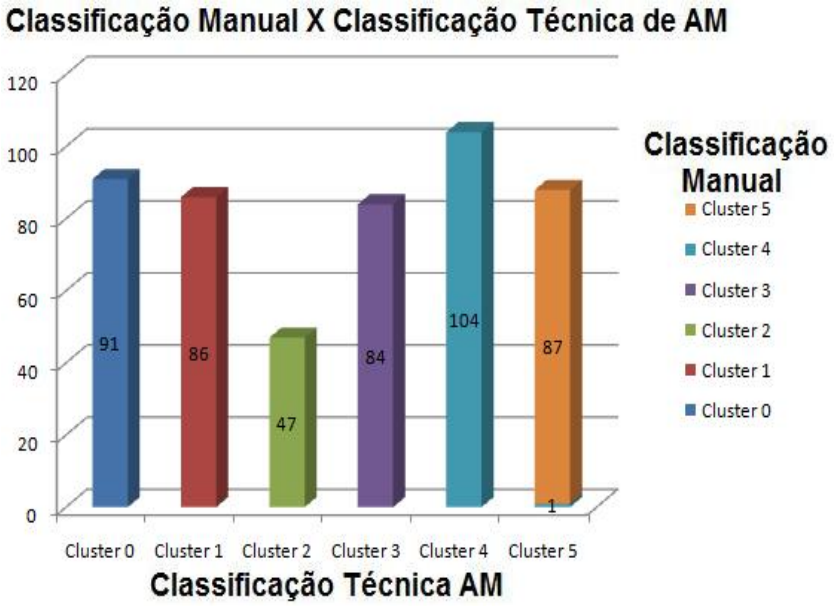


Figura 6.2: Gráfico comparativo de classes de equivalência para o programa *Triângulo* - Experimento 1.3.7

classe de equivalência 1 (valores desordenados) são classificados pelas técnicas de agrupamento como sendo da classe 2 (valores ordenados inversamente). Este fato é justificável, visto que as classes de equivalência definidas dependem do conhecimento do testador e, provavelmente, outro testador definiria outras classes de equivalência para o mesmo programa. Outro fato importante é que os valores ordenados inversamente não deixam de ser valores desordenados, o que também justifica a classificação obtida pelas técnicas de agrupamento.

O melhor resultado obtido dentre todos experimentos foi utilizando a técnica *COBWEB* no experimento 2.2.3 (atributos: número de mutantes mortos e porcentagem de cobertura das classes de defeitos). Para esta classificação percebe-se que somente 7 casos de teste não foram classificados corretamente e o resultado é apresentado na Figura 6.3. Já o resultado obtido com a técnica *K-Means* em 2.1.3 (atributos: número de mutantes mortos e porcentagem de cobertura das classes de defeitos) é apresentado na Figura 6.4 onde verifica-se que 46 casos de teste foram classificados incorretamente.

A Figura 6.5 apresenta os resultados para as técnicas *COBWEB* e *EM*, visto que as duas geram exatamente o mesmo resultado. O gráfico refere-se aos experimentos 2.2.6 e 2.3.6 sendo que os dois utilizam os mesmos atributos: porcentagens de cobertura dos

critérios estruturais, número de mutantes mortos e porcentagem de cobertura das classes de defeitos. Pôde-se analisar que somente 4 dos 21 experimentos geraram resultados bons ou satisfatórios. Nesses 4 experimentos o resultado final demonstra que alguns dos casos de teste referentes à classe de equivalência 1 (valores desordenados) são classificados pelas técnicas de agrupamento como sendo da classe 2 (valores ordenados inversamente).

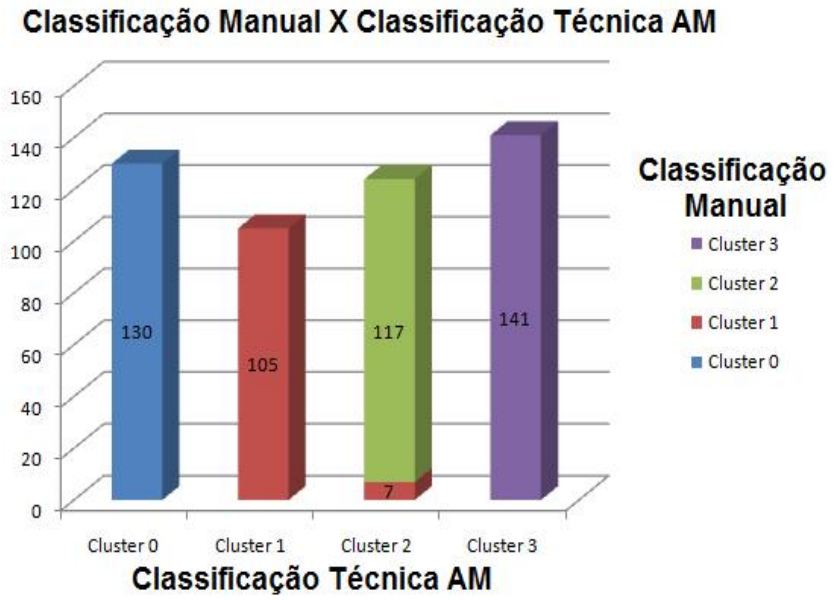


Figura 6.3: Gráfico comparativo de classes de equivalências para o programa *Bubble Sort* - Experimento 2.2.3

6.2.3 *FourBalls*

Assim como para os demais programas, foram utilizadas as três técnicas de agrupamento para realizar 12 experimentos. Desses, 6 geraram resultados bons: 3.2.5 (técnica: *COBWEB*), 3.3.5 (técnica: *EM*), 3.2.6 (técnica: *COBWEB*), 3.3.6 (técnica: *EM*), 3.2.7 (técnica: *COBWEB*) e 3.3.7 (técnica: *EM*). Relembrando que os experimentos que terminam pelo número 5 utilizam como atributos: entradas, saídas, número de mutantes mortos e porcentagem de cobertura das classes de defeitos; os que terminam pelo número 6 utilizam: porcentagens de cobertura dos critérios estruturais, número de mutantes mortos e porcentagem de cobertura das classes de defeitos; e os terminados pelo número 7 utilizam todos os atributos.

Nesses 6 experimentos, todos os casos de teste são classificados corretamente quanto

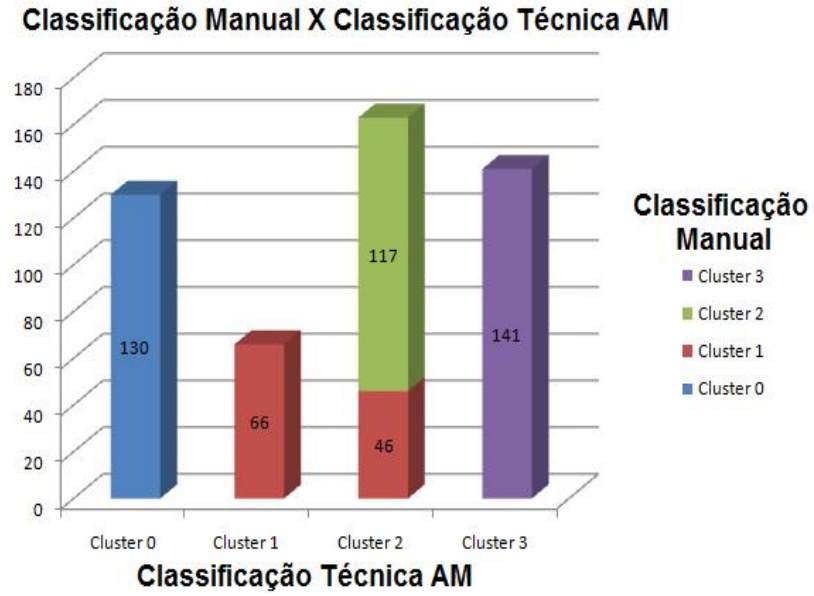


Figura 6.4: Gráfico comparativo de classes de equivalências para o programa *Bubble Sort* - Experimento 2.1.3

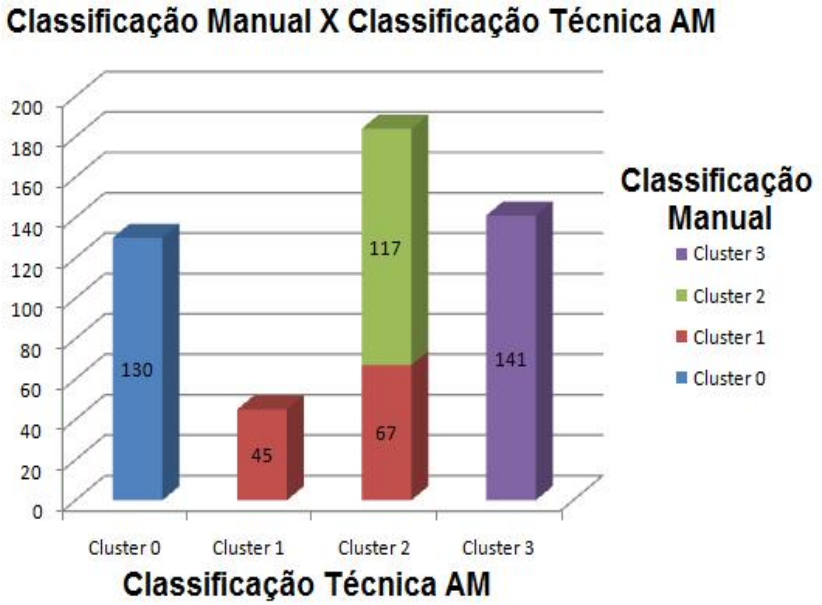


Figura 6.5: Gráfico comparativo de classes de equivalências para o programa *Bubble Sort* - Experimentos 2.2.6 e 2.3.6

às classes de equivalência. Na Figura 6.6 é apresentada a classificação obtida. Visto que os resultados referentes aos seis experimentos são idênticos, o gráfico apresentado corresponde a esses seis experimentos, independente da técnica e dos atributos de entrada.

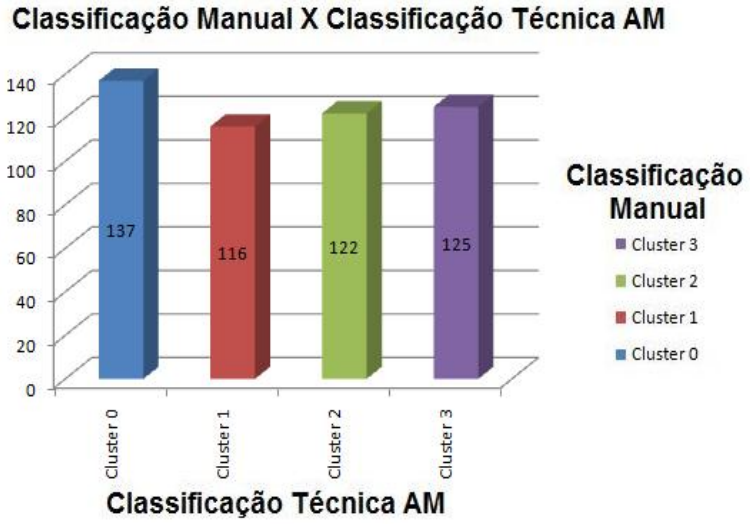


Figura 6.6: Gráfico comparativo de classes de equivalências para o programa *FourBalls* - Experimentos 3.2.5, 3.3.5, 3.2.6, 3.3.6, 3.2.7 e 3.3.7

6.2.4 *GetCmd*

Com relação ao programa *GetCmd* observa-se que dois dos três experimentos realizados geraram resultados bons, ou seja, os experimentos realizados utilizando todos os atributos de entrada com as técnicas *K-Means* e *COBWEB*, 4.1.7 e 4.2.7, respectivamente. Nesses dois experimentos o resultado final demonstra que as técnicas de agrupamento classificam todos os casos de teste corretamente em relação à classificação manual. Na Figura 6.7 é apresentada a classificação obtida.

6.2.5 Discussão sobre Identificação das Classes de Equivalência

De modo geral, a identificação de classes de equivalência pode ser considerada muito boa, visto que, para todos os programas utilizados nos experimentos, os algoritmos de agrupamento conseguiram identificar as classes de equivalência de maneira apropriada, ou seja, gerando resultados bons e satisfatórios.

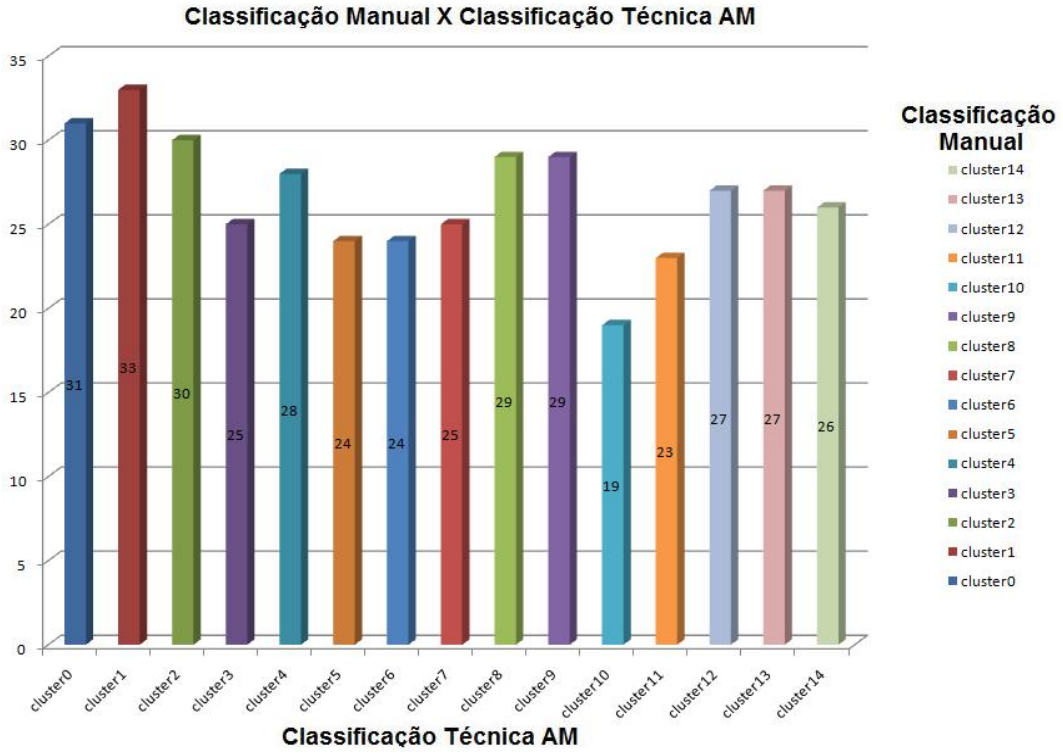


Figura 6.7: Gráfico comparativo de classes de equivalências para o programa *GetCmd* - Experimentos 4.1.7 e 4.2.7

Avaliando as técnicas de agrupamento utilizadas nos experimentos, pode-se concluir que não existe uma técnica melhor que outra. Cada técnica possui características específicas e possuem melhor desempenho em determinadas situações. Esse fato é comprovado ao analisar os melhores resultados obtidos para os programas, onde todas as técnicas foram boas ou satisfatórias. O algoritmo *K-means* gerou resultados bons ou satisfatórios em 3 casos, já o *COBWEB* foi bom ou satisfatório em 12 casos, enquanto o *EM* foi bom ou satisfatório em 8 casos.

Considerando os atributos de entrada para as técnicas de agrupamento, nota-se que a única combinação de atributos que não gerou resultados satisfatórios para nenhum dos programas foi a combinação que utiliza somente as entradas e saídas. Para o programa *Triângulo* todas as demais combinações geraram resultados bons ou satisfatórios. Já para o programa *Bubble Sort* as combinações que geraram resultados satisfatórios sempre envolvem o número de mutantes mortos e a porcentagem de cobertura das classes de defeitos, podendo também envolver a cobertura dos critérios estruturais. O programa *FourBalls* apresenta resultados bons quando utiliza as seguintes combinações de atributos:

entradas, saídas, número de mutantes mortos e porcentagem de cobertura das classes de defeitos; porcentagens de cobertura dos critérios estruturais, número de mutantes mortos e porcentagem de cobertura das classes de defeitos; e todos os atributos. Por fim, o programa *GetCmd* apresenta resultados bons somente utilizando todos os atributos de entrada.

Em relação aos parâmetros utilizados para cada uma das técnicas de agrupamento nos diferentes programas, observa-se que existe uma variação pequena. Somente o programa *GetCmd* que necessitou de parâmetros com valores bem diferentes dos demais programas para gerar uma classificação boa, conforme observado na Tabela 6.5.

Os programas utilizados nos experimentos foram programas onde a identificação de classes de equivalência de maneira manual se dá de forma simples. Foram escolhidos tais programas para facilitar a validação da identificação de classes pelas técnicas de AM em relação às classes definidas manualmente. Pôde-se notar que as melhores classificações foram obtidas para os programas *FourBalls* e *GetCmd*, onde todas os casos de teste foram classificados corretamente quanto às classes de equivalência. Já para os programas *Triângulo* e *Bubble Sort*, alguns casos de teste foram classificados diferentemente da classificação manual, o que não pode ser considerado incorreto, visto que a classificação gerada possui uma lógica. De forma geral, pode-se dizer que a escolha dos programas para os experimento foi boa, contudo, poderia ter sido utilizado algum programa mais complexo para ser avaliado também.

6.3 Geração de Regras

Utilizando a abordagem proposta, o classificador *J48* disponível na ferramenta *Weka* foi acionado para gerar regras sobre as informações obtidas da execução dos testes. Conforme explicado anteriormente, após a identificação das classes de equivalência, é possível associar, através de regras, os dados oriundos das técnicas estrutural e baseada em defeitos com as classes de equivalência identificadas, referentes à técnica funcional. Tal relacionamento entre as informações da atividade de teste, expresso através de regras, é utilizado para apoio ao teste de regressão. Essas regras possibilitam a redução, seleção e priorização

de casos de teste. Para os melhores resultados obtidos em relação à identificação de classes de equivalência, apresentados na Seção 6.2, foi executado o classificador *J48* e a seguir são apresentadas as regras geradas para tais experimentos.

Para o programa *Triângulo*, os experimentos 1.1.2, 1.2.2 e 1.3.2, ou seja, os experimentos utilizando as três técnicas de agrupamento e como atributos as porcentagens de cobertura dos critérios estruturais, geraram exatamente as mesmas regras. Tais regras são apresentadas em Regras 6. 1. Nota-se que as regras utilizaram como base somente dois atributos: cobertura do critério Todos-Nós e cobertura do critério PU.

```
coberturaNos <= 38.888889
|   coberturaNos <= 16.666667: cluster0 (91.0)
|   coberturaNos > 16.666667
|   |   coberturaPU <= 11.428572: cluster1 (86.0)
|   |   coberturaPU > 11.428572: cluster3 (84.0)
coberturaNos > 38.888889
|   coberturaPU <= 16.428571: cluster2 (47.0)
|   coberturaPU > 16.428571: cluster4 (192.0)
```

Regras 6. 1: Regras geradas para o programa *Triângulo* - Experimentos 1.1.2, 1.2.2 e 1.3.2

Já o experimento 1.2.3 para o programa *Triângulo*, o qual foi conduzido utilizando a técnica *COBWEB* e como atributos o número de mutantes mortos e a porcentagem de cobertura das classes de defeitos, gerou as regras apresentadas em Regras 6. 2. Percebe-se que as regras utilizaram como base somente duas classes de defeitos como atributos: STRI (é projetado para garantir que a condição de cada comando *if* seja avaliado pelo menos uma vez pelo ramo verdadeiro e pelo ramo falso) e SRSR (troca cada comando de uma função por todos os comando *return* que existem na mesma função).

O experimento 1.2.4 para o programa *Triângulo*, que também foi conduzido utilizando a técnica *COBWEB* tendo como atributos as entradas, saídas e porcentagem de cobertura dos critérios estruturais, gerou as regras apresentadas em Regras 6. 3. Nota-se que as regras utilizaram como base somente três atributos: cobertura do critério Todos-Nós, cobertura do critério PU e saída 1.

Os experimentos 1.2.5 e 1.3.5, para o programa *Triângulo*, foram conduzidos utilizando as técnicas *COBWEB* e *EM*, respectivamente. Como atributos para os dois experimen-

```
STRI <= 25
|   STRI <= 8.333333: cluster0 (91.0)
|   STRI > 8.333333
|   |   SRSR <= 37.5: cluster1 (86.0)
|   |   SRSR > 37.5: cluster3 (84.0)
STRI > 25
|   SRSR <= 47.5: cluster2 (47.0)
|   SRSR > 47.5: cluster4 (192.0)
```

Regras 6. 2: Regras geradas para o programa *Triângulo* - Experimento 1.2.3

```
coberturaNos <= 38.888889
|   coberturaNos <= 16.666667: cluster0 (91.0)
|   coberturaNos > 16.666667
|   |   saida1 <= 2: cluster1 (86.0)
|   |   saida1 > 2: cluster3 (84.0)
coberturaNos > 38.888889
|   coberturaPU <= 16.428571: cluster2 (47.0)
|   coberturaPU > 16.428571: cluster4 (192.0)
```

Regras 6. 3: Regras geradas para o programa *Triângulo* - Experimento 1.2.4

tos foram utilizados: entradas, saídas, número de mutantes mortos e porcentagem de cobertura das classes de defeitos. Estes experimentos geraram as regras apresentadas em Regras 6. 4. Percebe-se que as regras utilizaram como base somente três atributos: a saída 1 e as classes de defeitos STRI (é projetado para garantir que a condição de cada comando *if* seja avaliado pelo menos uma vez pelo ramo verdadeiro e pelo ramo falso) e SRSR (troca cada comando de uma função por todos os comando *return* que existem na mesma função).

```
STRI <= 25
|   STRI <= 8.333333: cluster0 (91.0)
|   STRI > 8.333333
|   |   saida1 <= 2: cluster1 (86.0)
|   |   saida1 > 2: cluster3 (84.0)
STRI > 25
|   SRSR <= 47.5: cluster2 (47.0)
|   SRSR > 47.5: cluster4 (192.0)
```

Regras 6. 4: Regras geradas para o programa *Triângulo* - Experimentos 1.2.5 e 1.3.5

Por fim, as regras geradas para os experimentos 1.2.7 e 1.3.7 são apresentadas em Regras 6. 5. Os experimentos 1.2.7 e 1.3.7, para o programa *Triângulo*, utilizam todos os

atributos de entrada e foram conduzidos utilizando as técnicas *COBWEB* e *EM*, respectivamente. Nota-se que as regras utilizaram como base somente três atributos: cobertura do critério Todos-Nós, cobertura do critério PU e saída 1.

```
coberturaNos <= 38.888889
|   coberturaNos <= 16.666667: cluster0 (91.0)
|   coberturaNos > 16.666667
|   |   saida1 <= 2: cluster1 (86.0)
|   |   saida1 > 2: cluster3 (84.0)
coberturaNos > 38.888889
|   coberturaPU <= 16.428571: cluster2 (47.0)
|   coberturaPU > 16.428571
|   |   saida1 <= 4: cluster4 (105.0/1.0)
|   |   saida1 > 4: cluster5 (87.0)
```

Regras 6. 5: Regras geradas para o programa *Triângulo* - Experimentos 1.2.7 e 1.3.7

O experimento 2.1.3, para o programa *Bubble Sort*, foi conduzido utilizando a técnica *K-means* e como atributos foram utilizados: número de mutantes mortos e porcentagem de cobertura das classes de defeitos. Este experimento gerou as regras apresentadas em Regras 6. 6. Percebe-se que as regras utilizaram como base sete classes de defeitos: SRSR (Troca cada comando de uma função por todos os comando *return* que existem na mesma função), SSDL (É projetado para mostrar que cada comando do programa tem um efeito na saída do mesmo. Sendo assim, quando aplicado, ele deleta cada um dos comandos do programa para verificar se afeta a saída), OASN (Troca um operador aritmético sem atribuições por um operador de deslocamento sem atribuições), Cccr (Troca constantes por todas as constantes do programa), SMVB (Modela defeitos de uso incorreto de final de comandos, ou seja, uso incorreto de (})), VDTR (Requer valores negativos, positivos e zero para cada referência escalar) e Ccsr (Troca referências escalares por constantes).

Já o experimento 2.2.3, para o programa *Bubble Sort*, foi conduzido utilizando a técnica *COBWEB* e como atributos também foram utilizados: número de mutantes mortos e porcentagem de cobertura das classes de defeitos. Este experimento gerou as regras apresentadas em Regras 6. 7. Nessas regras percebe-se que também foi utilizado como base o número de mutantes mortos como atributo, além das classes de defeitos SRSR (Troca cada comando de uma função por todos os comando *return* que existem na mesma

```
SRSR <= 88
|   SSDL <= 56: cluster3 (141.0)
|   SSDL > 56: cluster0 (130.0)
SRSR > 88
|   OASN <= 62.5
|   |   Cccr <= 85
|   |   |   SMVB <= 85.714286: cluster1 (2.0)
|   |   |   SMVB > 85.714286: cluster2 (4.0)
|   |   Cccr > 85: cluster2 (131.0)
|   OASN > 62.5
|   |   OASN <= 75
|   |   |   SMVB <= 85.714286: cluster1 (14.0)
|   |   |   SMVB > 85.714286
|   |   |   |   VDTR <= 48
|   |   |   |   |   Ccsr <= 94.666667: cluster1 (3.0)
|   |   |   |   |   Ccsr > 94.666667: cluster2 (27.0)
|   |   |   |   VDTR > 48: cluster1 (10.0/1.0)
|   |   OASN > 75: cluster1 (38.0)
```

Regras 6. 6: Regras geradas para o programa *Bubble Sort* - Experimento 2.1.3

função), SSDL (É projetado para mostrar que cada comando do programa tem um efeito na saída do mesmo. Sendo assim, quando aplicado, ele deleta cada um dos comandos do programa para verificar se afeta a saída), OASN (Troca um operador aritmético sem atribuições por um operador de deslocamento sem atribuições), e VTWD (Troca referência escalar pelo seu valor sucessor e predecesssor).

```
SRSR <= 88
|   SSDL <= 56: cluster3 (141.0)
|   SSDL > 56: cluster0 (130.0)
SRSR > 88
|   VTWD <= 74
|   |   nroMutantesMortos <= 772: cluster1 (7.0)
|   |   nroMutantesMortos > 772
|   |   |   OASN <= 62.5: cluster2 (123.0)
|   |   |   OASN > 62.5: cluster1 (3.0)
|   VTWD > 74: cluster1 (96.0/1.0)
```

Regras 6. 7: Regras geradas para o programa *Bubble Sort* - Experimento 2.2.3

Os experimentos 2.2.6 e 2.3.6, para o programa *Bubble Sort*, utilizam como base os atributos: porcentagem de cobertura dos critérios estruturais, número de mutantes mortos e porcentagem de cobertura das classes de defeitos e foram conduzidos utilizando as

técnicas *COBWEB* e *EM*. Estes experimentos geraram as regras apresentadas em Regras 6. 8. Percebe-se que as regras utilizaram como base quatro atributos: porcentagem de cobertura dos critérios Todos-Arcos e PDU, número de mutantes mortos e a classe de defeitos SSDL (É projetado para mostrar que cada comando do programa tem um efeito na saída do mesmo. Sendo assim, quando aplicado, ele deleta cada um dos comandos do programa para verificar se afeta a saída).

```
coberturaArcs <= 83.333333
|   SSDL <= 56: cluster3 (141.0)
|   SSDL > 56: cluster0 (130.0)
coberturaArcs > 83.333333
|   nroMutantesMortos <= 770: cluster1 (37.0)
|   nroMutantesMortos > 770
|   |   coberturaPDU <= 52.499998: cluster1 (8.0)
|   |   coberturaPDU > 52.499998: cluster2 (184.0)
```

Regras 6. 8: Regras geradas para o programa *Bubble Sort* - Experimentos 2.2.6 e 2.3.6

Os experimentos 3.2.5 e 3.3.5, para o programa *FourBalls*, foram conduzidos utilizando as técnicas *COBWEB* e *EM*, respectivamente. Como atributos para os dois experimentos foram utilizados: entradas, saídas, número de mutantes mortos e porcentagem de cobertura das classes de defeitos. Estes experimentos geraram as regras apresentadas em Regras 6. 9. Percebe-se que as regras utilizaram como base somente dois atributos: a entrada 1 e a classe de defeitos SRSR (Troca cada comando de uma função por todos os comando *return* que existem na mesma função).

```
SRSR <= 56.666667
|   entrada1 <= -100: cluster0 (137.0)
|   entrada1 > -100: cluster1 (116.0)
SRSR > 56.666667
|   entrada1 <= 100: cluster2 (122.0)
|   entrada1 > 100: cluster3 (125.0)
```

Regras 6. 9: Regras geradas para o programa *FourBalls* - Experimentos 3.2.5 e 3.3.5

Os experimentos 3.2.6 e 3.3.6, para o programa *FourBalls*, utilizam como base os atributos: porcentagem de cobertura dos critérios estruturais, número de mutantes mortos e porcentagem de cobertura das classes de defeitos e foram conduzidos utilizando as técnicas

COBWEB e *EM*. Estes experimentos geraram as regras apresentadas em Regras 6. 10. Percebe-se que as regras utilizaram como base dois atributos: porcentagem de cobertura do critério Todos-Nós e a classe de defeitos ORAN (Troca um operador relacional por um operador aritmético sem atribuições).

```
coberturaNos <= 61.538462
|   coberturaNos <= 46.153846: cluster0 (137.0)
|   coberturaNos > 46.153846: cluster1 (116.0)
coberturaNos > 61.538462
|   ORAN <= 65: cluster2 (122.0)
|   ORAN > 65: cluster3 (125.0)
```

Regras 6. 10: Regras geradas para o programa *FourBalls* - Experimentos 3.2.6 e 3.3.6

Por fim, os experimentos 3.2.7 e 3.3.7 conduzidos para o programa *FourBalls*, utilizando todos os atributos de entrada e as técnicas *COBWEB* e *EM*, respectivamente, são apresentadas em Regras 6. 11. Nota-se que as regras utilizaram como base somente dois atributos: cobertura do critério Todos-Nós e entrada 1.

```
coberturaNos <= 61.538462
|   entrada1 <= -100: cluster0 (137.0)
|   entrada1 > -100: cluster1 (116.0)
coberturaNos > 61.538462
|   entrada1 <= 100: cluster2 (122.0)
|   entrada1 > 100: cluster3 (125.0)
```

Regras 6. 11: Regras geradas para o programa *FourBalls* - Experimentos 3.2.7 e 3.3.7

Para o programa *GetCmd*, os experimentos 4.1.7 e 4.2.7 conduzidos utilizando todos os atributos de entrada e as técnicas *K-means* e *COBWEB*, respectivamente, são apresentadas em Regras 6. 12. Nota-se que as regras utilizaram como base três atributos: a entrada 1, a saída 1 e a porcentagem de cobertura do critério Todos-Nós.

6.3.1 Discussão sobre Geração das Regras

As regras geradas pelo classificador *J48* dependem exclusivamente da forma como as classes de equivalência são geradas pelas técnicas de agrupamento. Ou seja, se as técnicas de agrupamento utilizam entradas e saídas para identificar os *clusters*, as regras geradas

```
coberturaNos <= 20.930233
|   entrada1 <= 96
|   |   saida1 <= 6
|   |   |   entrada1 <= 94: cluster5 (24.0)
|   |   |   entrada1 > 94: cluster4 (28.0)
|   |   saida1 > 6
|   |   |   entrada1 <= 95: cluster6 (24.0)
|   |   |   entrada1 > 95: cluster7 (25.0)
|   entrada1 > 96
|   |   saida1 <= 5
|   |   |   entrada1 <= 98: cluster2 (30.0)
|   |   |   entrada1 > 98: cluster0 (31.0)
|   |   saida1 > 5: cluster1 (33.0)
coberturaNos > 20.930233
|   coberturaNos <= 30.232558
|   |   entrada1 <= 90
|   |   |   entrada1 <= 89: cluster10 (19.0)
|   |   |   entrada1 > 89: cluster9 (29.0)
|   |   entrada1 > 90
|   |   |   entrada1 <= 91: cluster8 (29.0)
|   |   |   entrada1 > 91: cluster3 (25.0)
|   coberturaNos > 30.232558
|   |   coberturaNos <= 34.883721
|   |   |   entrada1 <= 87: cluster12 (27.0)
|   |   |   entrada1 > 87: cluster11 (23.0)
|   |   coberturaNos > 34.883721
|   |   |   entrada1 <= 85: cluster14 (26.0)
|   |   |   entrada1 > 85: cluster13 (27.0)
```

Regras 6. 12: Regras geradas para o programa *GetCmd* - Experimentos 4.1.7 e 4.2.7

serão obtidas com base nas entradas e saídas. Sendo assim, nem sempre são geradas regras que são úteis para o teste de regressão.

O único parâmetro de configuração do algoritmo *J48* que foi alterado em relação às configurações padrões, foi o parâmetro relacionado à poda da árvore de decisão. Foi considerada a árvore sem poda, visto que as regras geradas com a poda foram ainda mais simples que as regras sem a poda.

As regras obtidas foram boas. Contudo, a utilização de outros algoritmos geradores de regras também poderia ser estudada. Neste caso poderia ser realizada uma comparação entre diferentes classificadores e entre a qualidade das regras geradas por cada um deles.

6.4 Aplicação das Regras

A abordagem proposta classifica os casos de teste em classes de equivalência através das informações de execução dos testes. Essa classificação pode ser utilizada para redução de um conjunto de casos de teste. Uma abordagem comum propõe a execução do teste de regressão escolhendo pelo menos um caso de teste de cada classe. Sendo assim, considerando como exemplo o programa *Triângulo*, somente 5 casos de teste seriam selecionados. Outra abordagem consiste em selecionar um número x de casos de teste com maior cobertura estrutural. Considerando x igual a 5, têm-se os cinco melhores casos de teste de cada classe, ou seja, 25 casos de teste. Sabe-se que o problema de reduzir um conjunto de casos de teste é a eficácia. Para garantir a detecção de defeitos específicos, o tipo do mutante (ou operador de mutação) coberto por cada caso de teste pode ser considerado na escolha dos casos de teste de cada classe.

Utilizando as regras geradas nesse estudo, o testador pode identificar características específicas de cada classe de equivalência, percebendo que algumas classes podem ser melhores que outras, dependendo do aspecto que o testador deseja avaliar ou priorizar no teste de regressão. Por exemplo, um caso de teste de uma classe pode ter porcentagem de cobertura estrutural maior que todas as outras classes e também cobrir uma determinada classe de defeitos, enquanto os casos de teste de outras classes não cobrem. Esse tipo de informação é útil para ajudar o testador a identificar os melhores casos de teste para o

teste de regressão. O testador pode identificar as classes mais importantes, ou seja, classes que cobrem partes críticas do programa ou que geram maior cobertura de um determinado critério de teste, priorizando os casos de teste destas classes, ou selecionando mais casos de teste destas classes. Essas são algumas possibilidades de aplicação das regras, dentre outras possíveis.

A aplicação das regras seguiu uma abordagem baseada em progressão geométrica (PG), onde cada elemento da PG corresponde à porcentagem de casos de teste que deve ser selecionada para cada classe de equivalência. Inicialmente as classes de equivalência são ordenadas conforme o critério enfatizado, em seguida, os casos de teste são selecionados com base na porcentagem gerada para o elemento da PG. Assume-se que para a classe de equivalência com menor prioridade será selecionado somente um caso de teste. A partir daí, os elementos da PG indicam a quantidade de casos de teste que devem ser selecionados para cada classe. A seguir, para cada programa foi definida uma forma de aplicação das regras para gerar um conjunto de casos de teste para o teste de regressão. Também foram avaliados os conjuntos de casos de teste gerados com relação à abordagem padrão onde é selecionado um caso de teste de cada classe de equivalência e também em relação à abordagem *retest-all*. Outra aplicação, também demonstrada nessa seção, refere-se à introdução de um novo caso de teste ao conjunto de casos de teste de regressão.

6.4.1 Aplicação das Regras para o Programa *Triângulo*

Assumindo que a classe de defeitos que o testador deseja enfatizar para o teste de regressão do programa *Triângulo* é *STRI* (o qual é projetado para garantir que a condição de cada comando *if* seja avaliado pelo menos uma vez pelo ramo verdadeiro e pelo ramo falso) e analisando as regras geradas pelo experimento 1.2.3, pode-se afirmar que é mais vantajoso selecionar casos de teste do *cluster4*, porque os casos de teste dele geram porcentagem de cobertura maior para esta classe de defeitos. Também percebe-se que selecionar casos de teste do *cluster0* é menos vantajoso que selecionar casos de teste de outros *clusters*. Isso ocorre porque os casos de teste desse *cluster* geram menor porcentagem de cobertura para esta classe de defeitos.

Utilizando a abordagem baseada em PG para gerar a porcentagem de casos de teste de cada classe, utiliza-se valor inicial a igual a $0,011$. Este valor inicial foi encontrado assumindo que a quantidade mínima de casos de teste selecionados para a classe de equivalência com menor prioridade é 1, *cluster0* nesse caso. O *cluster0* contém 91 casos de teste e para selecionar 1 caso de teste desta classe, o valor inicial deve ser $0,011$. Assumindo que o testador quer enfatizar a cobertura da classe de defeitos STRI, o teste pode seguir a ordem de prioridade estabelecida para os *clusters*: *cluster4, cluster2, cluster3, cluster1 e cluster0*. A razão para esta PG foi calculada utilizando o valor inicial, o valor final fixo em 10% sobre o número total de casos de teste e o número de classes de equivalência (5). Sendo assim, a razão é igual a $1,74$ e a porcentagem de casos de teste para cada classe está apresentada na Tabela 6.6.

A ordem de prioridade mencionada anteriormente também depende do aspecto que o testador considera mais relevante para o teste de regressão. Assumindo novamente que o testador deseja enfatizar a cobertura da classe de defeitos *STRI* para o programa *Triângulo* e analisando as regras geradas, pode se dizer que a ordem de execução dos casos de teste deve seguir a ordem das classes de equivalência: *cluster4, cluster2, cluster3, cluster1 e cluster0*. Ao executar os casos de teste dessas classes de equivalência utilizando a ordem de prioridade estabelecida, é possível obter cobertura da classe de defeitos *STRI* mais rapidamente.

Para este programa em específico, independentemente do critério a ser enfatizado (Todos-Nós, PU ou classes de defeitos), percebe-se, ao analisar as regras, que a ordem das classes de equivalência não é alterada: *cluster4, cluster2, cluster3, cluster1 e cluster0*. Sendo assim, esse conjunto de casos de teste reduzido também obtém cobertura dos critérios Todos-Nós e PU mais rapidamente, além de cobrir mais rapidamente a classe de defeitos *SRSR* (Troca cada comando de uma função por todos os comando *return* que existem na mesma função).

A redução proposta gerou T' com 28 enquanto a abordagem convencional gerou T' com 5 casos de teste. A seguir, foi conduzida uma comparação entre a abordagem utilizando PG, a abordagem padrão e a abordagem *retest-all*. Esta comparação é apresentada

Tabela 6.6: Número de casos de teste selecionados na redução para o programa *Triângulo*

Classe de Equivalência	Porcentagem	# Casos de Teste
cluster0	0,011	1
cluster1	0,019	2
cluster3	0,033	3
cluster2	0,058	3
cluster4	0,1	19
Total		28

na Tabela 6.7 onde pôde-se observar que a abordagem utilizando PG garante a mesma cobertura para os critérios estruturais que a abordagem *retest-all* e cobertura superior se comparado com a abordagem padrão. Ainda percebe-se que a única disparidade entre essas duas abordagens é referente ao número de mutantes mortos onde a abordagem *retest-all* matou 3 mutantes a mais.

Tabela 6.7: Comparação das abordagens para o programa *Triângulo*

	Retest-All	Padrão	Abordagem PG
Cobertura Nós	100	77,77	100
Cobertura Arcos	85,71	57,14	85,71
Cobertura PU	90	65	90
Cobertura PDU	90	65	90
Cobertura PUDU	90	65	90
Score de Mutação	0,83	0,57	0,83
Mutantes Mortos	2059	1422	2056

6.4.2 Aplicação das Regras para o Programa *Bubble Sort*

Para este programa as regras foram aplicadas de duas maneiras: 1) enfatizando critérios estruturais (Todos-Arcos e PU) e 2) enfatizando cobertura de classes de defeitos (*SRSR* (Troca cada comando de uma função por todos os comando *return* que existem na mesma função) e *VTWD* (Troca referência escalar pelo seu valor sucessor e predecesssor)). A seguir cada uma das aplicações são apresentadas.

6.4.2.1 Aplicação das Regras com Ênfase nos Critérios Estruturais

Assumindo que os critérios estruturais que o testador deseja enfatizar para o teste de regressão do programa *Bubble Sort* são Todos-Arcos e PU e analisando as regras geradas para os experimentos 2.2.6 e 2.3.6, pode-se estabelecer a seguinte ordem de prioridade para as classes de equivalência: *cluster2, cluster1, cluster0 e cluster3.*

Mais uma vez a porcentagem de casos de teste de cada classe que será selecionada para o teste de regressão foi gerada através da abordagem utilizando PG. Esta PG possui valor inicial a igual a $0,007$, o qual foi calculado a partir do cluster3 (classe com menor prioridade) da qual deve ser selecionado apenas 1 caso de teste. O cluster3 contém 141 casos de teste e para selecionar 1 caso de teste desta classe, o valor inicial deve ser $0,007$. A razão para esta PG foi calculada utilizando o valor inicial, o valor final fixo em 10% sobre o número total de casos de teste e o número de classes de equivalência (4). Sendo assim, a razão é igual a $2,43$ e a porcentagem de casos de teste para cada classe está apresentada na Tabela 6.8.

Tabela 6.8: Número de casos de teste selecionados na redução com ênfase nos critérios estruturais para o programa *Bubble Sort*

Classe de Equivalência	Porcentagem	# Casos de Teste
cluster3	0,007	1
cluster0	0,017	2
cluster1	0,04	5
cluster2	0,1	12
Total		20

A redução proposta gerou T' com 20 enquanto a abordagem convencional gerou T' com 4 casos de teste. Foi conduzida uma comparação entre a abordagem utilizando PG, a abordagem convencional e a abordagem *retest-all*. Esta comparação é apresentada na Tabela 6.9 onde pôde-se observar que as três abordagens obtiveram exatamente o mesmo resultado. Nesse caso a identificação de classes de equivalência fez com que o conjunto de casos de teste original pudesse ser reduzido de 500 casos de teste para apenas 4 casos de teste, garantindo a mesma cobertura estrutural e de classes de defeitos.

Tabela 6.9: Comparação das abordagens para o programa *Bubble Sort* com ênfase nos critérios estruturais

	Retest-All	Padrão	Abordagem PG
Cobertura Nós	100	100	100
Cobertura Arcos	100	100	100
Cobertura PU	70,37	70,37	70,37
Cobertura PDU	55,88	55,88	55,88
Cobertura PUDU	53,70	53,70	53,70
Score de Mutação	0,9	0,9	0,9
Mutantes Mortos	793	793	793

6.4.2.2 Aplicação das Regras com Ênfase nas Classes de Defeitos

Assumindo que as classes de defeitos que o testador deseja enfatizar para o teste de regressão do programa *Bubble Sort* são *SRSR* e *VTWD* e analisando as regras geradas para o experimento 2.2.3, pode-se estabelecer a seguinte ordem de prioridade para as classes de equivalência: *cluster1, cluster2, cluster0 e cluster3.*

Novamente a porcentagem de casos de teste de cada classe que será selecionada para o teste de regressão foi gerada através da abordagem utilizando PG. Como a classe com menor prioridade continua sendo *cluster3*, o valor inicial também deve ser $0,007$ e a razão é igual a $2,43$ e a porcentagem de casos de teste para cada classe está apresentada na Tabela 6.10.

Tabela 6.10: Número de casos de teste selecionados na redução com ênfase nas classes de defeitos para o programa *Bubble Sort*

Classe de Equivalência	Porcentagem	# Casos de Teste
cluster3	0,007	1
cluster0	0,017	2
cluster2	0,04	5
cluster1	0,1	11
Total		19

A redução proposta gerou T' com 19 enquanto a abordagem convencional gerou T' com 4 casos de teste. Foi conduzida uma comparação entre a abordagem utilizando PG, a abordagem convencional e a abordagem *retest-all.* Esta comparação é apresentada

na Tabela 6.11 onde percebe-se exatamente o mesmo resultado gerado enfatizando os critérios estruturais, ou seja, as três abordagens obtiveram exatamente o mesmo resultado, possibilitando uma redução de 500 casos de teste para apenas 4 casos de teste e garantindo a mesma cobertura estrutural e de classes de defeitos.

Tabela 6.11: Comparação das abordagens para o programa *Bubble Sort* com ênfase nas classes de defeitos

	Retest-All	Padrão	Abordagem PG
Cobertura Nós	100	100	100
Cobertura Arcos	100	100	100
Cobertura PU	70,37	70,37	70,37
Cobertura PDU	55,88	55,88	55,88
Cobertura PUDU	53,70	53,70	53,70
Score de Mutação	0,9	0,9	0,9
Mutantes Mortos	793	793	793

6.4.3 Aplicação das Regras para o Programa *FourBalls*

Para este programa, baseando-se em todas as regras geradas, a ordem de prioridade das classes de equivalência sempre permanece: *cluster3, cluster2, cluster1 e cluster0*, independente da ênfase utilizada. Pode-se dizer que com essa ordem de prioridade enfatiza-se a cobertura do critério Todos-Nós e das classes de defeitos *SRSR* (Troca cada comando de uma função por todos os comando *return* que existem na mesma função) e *ORAN* (Troca um operador relacional por um operador aritmético sem atribuições).

A porcentagem de casos de teste de cada classe que será selecionada para o teste de regressão foi gerada através da abordagem utilizando PG. Esta PG possui valor inicial a igual a $0,0073$, o qual foi calculado a partir do *cluster0* (classe com menor prioridade) da qual deve ser selecionado apenas 1 caso de teste. O *cluster0* contém 137 casos de teste e para selecionar 1 caso de teste desta classe, o valor inicial deve ser $0,0073$. A razão para esta PG foi calculada utilizando o valor inicial, o valor final fixo em 10% sobre o número total de casos de teste e o número de classes de equivalência (4). Sendo assim, a razão é igual a $2,4$ e a porcentagem de casos de teste para cada classe está apresentada

na Tabela 6.12.

Tabela 6.12: Número de casos de teste selecionados na redução para o programa *FourBalls*

Classe de Equivalência	Porcentagem	# Casos de Teste
cluster0	0,0073	1
cluster1	0,017	2
cluster2	0,04	5
cluster3	0,1	13
Total		21

A redução proposta gerou T' com 21 enquanto a abordagem convencional gerou T' com 4 casos de teste. Foi conduzida uma comparação entre a abordagem utilizando PG, a abordagem convencional e a abordagem *retest-all*. Esta comparação é apresentada na Tabela 6.13 onde pôde-se observar que a abordagem padrão garante a mesma cobertura para os critérios estruturais que a abordagem utilizando PG e que a abordagem *retest-all*. A única diferença entre as três abordagens é o número de mutantes mortos onde a abordagem padrão mata um mutante a menos.

Tabela 6.13: Comparação das abordagens para o programa *FourBalls*

	Retest-All	Padrão	Abordagem PG
Cobertura Nós	100	100	100
Cobertura Arcos	100	100	100
Cobertura PU	95,65	95,65	95,65
Cobertura PDU	55	55	55
Cobertura PUDU	73,91	73,91	73,91
Score de Mutação	0,93	0,93	0,93
Mutantes Mortos	927	926	927

6.4.4 Aplicação das Regras para o Programa *GetCmd*

Para este programa, baseando-se nas regras geradas, a ordem de prioridade das classes de equivalência é: *cluster13, cluster14, cluster11, cluster12, cluster3, cluster8, cluster9, cluster10, cluster1, cluster0, cluster2, cluster7, cluster6, cluster4 e cluster5*, priorizando-se pelo critério Todos-Nós (único disponível nas regras).

A porcentagem de casos de teste de cada classe que será selecionada para o teste de regressão foi gerada através da abordagem utilizando PG. Esta PG possui valor inicial a igual a $0,041$, o qual foi calculado a partir do *cluster5* (classe com menor prioridade) da qual deve ser selecionado apenas 1 caso de teste. O *cluster5* contém 24 casos de teste e para selecionar 1 caso de teste desta classe, o valor inicial deve ser $0,041$. A razão para esta PG foi calculada utilizando o valor inicial, o valor final fixo em 50% sobre o número total de casos de teste e o número de classes de equivalência (15). Sendo assim, a razão é igual a $1,19$ e a porcentagem de casos de teste para cada classe está apresentada na Tabela 6.14.

Tabela 6.14: Número de casos de teste selecionados na redução para o programa *GetCmd*

Classe de Equivalência	Porcentagem	# Casos de Teste
cluster5	0,041	1
cluster4	0,05	1
cluster6	0,06	1
cluster7	0,07	2
cluster2	0,08	2
cluster0	0,1	3
cluster1	0,12	4
cluster10	0,14	3
cluster9	0,16	5
cluster8	0,2	6
cluster3	0,23	6
cluster12	0,28	8
cluster11	0,33	8
cluster14	0,4	10
cluster13	0,5	14
Total		74

A redução proposta gerou T' com 74 enquanto a abordagem convencional gerou T' com 15 casos de teste. Foi conduzida uma comparação entre a abordagem utilizando PG, a abordagem convencional e a abordagem *retest-all*. Esta comparação é apresentada na Tabela 6.15 onde pôde-se observar que as três abordagens obtiveram coberturas idênticas.

Tabela 6.15: Comparação das abordagens para o programa *GetCmd*

	Retest-All	Padrão	Abordagem PG
Cobertura Nós	69,76	69,76	69,76
Cobertura Arcos	93,75	93,75	93,75
Score de Mutação	0,41	0,41	0,41
Mutantes Mortos	507	507	507

6.4.5 Inclusão de um Novo Caso de Teste em T'

Outro uso para as regras obtidas pela abordagem é a classificação de um novo caso de teste durante o teste de regressão. Se o testador sabe a classe de equivalência do novo caso de teste, ele poderá saber de antemão algumas características desse caso de teste: limiar de cobertura para critérios estruturais, classes de defeitos relacionadas, etc. Com essas informações o testador pode avaliar o caso de teste utilizando as regras geradas para decidir se sua inclusão nesse conjunto irá agregar valor ao teste de regressão.

Assumindo a introdução do novo caso de teste t para o conjunto de testes de regressão T' onde: $T' = \{TC1 \in cluster0, TC2 \in cluster2, TC3 \in cluster2, TC4 \in cluster4, TC5 \in cluster4\}$ e t tem duas possibilidades, pertencer ao $cluster1$ ou ao $cluster3$; é necessário decidir qual *cluster* agrega mais valor ao teste de regressão. Assumindo que o testador deseja maximizar a cobertura do critério PU para o programa *Triângulo*. Através do exame das Regras 6. 1, nota-se que selecionar um caso de teste do $cluster3$ agrega mais valor ao conjunto T', porque os casos de teste desse *cluster* possuem porcentagem de cobertura do critério PU maior se comparados aos casos de teste do *cluster1*.

6.4.6 Discussão sobre Utilização das Regras

As regras geradas pelo classificador *J48* se mostraram úteis em alguns casos. Como no experimento realizado para o programa *Triângulo*, onde o conjunto de casos de teste selecionado através da abordagem utilizando PG gerou cobertura equivalente à abordagem *retest-all*, enquanto a abordagem padrão gerou resultados piores. Nos demais experi-

mentos utilizando PG, notou-se que os resultados se mostraram muito bons, contudo, os resultados utilizando a abordagem padrão, onde é selecionado um caso de teste de cada classe de equivalência, se mostraram tão bons quanto.

Visto que os programas utilizados nos experimentos são programas simples, percebe-se que as regras geradas também são simples. Para programas mais complexos, as regras possivelmente seriam mais expressivas. Tal fato percebe-se analisando os resultados do programa *Triângulo* que é o mais complexo dos programas utilizados e onde a aplicação das regras gerou melhores resultados.

Os resultados obtidos com as regras foram muito bons. Contudo, são necessários mais experimentos para avaliar o quanto as regras podem ser úteis ao teste de regressão, principalmente utilizando programas mais complexos onde as classes de equivalência não sejam tão simples de definir manualmente.

6.5 Considerações Finais

Este capítulo apresentou os principais resultados obtidos no trabalho realizado. Foram apresentados os resultados relacionados à identificação de classes de equivalência, em relação à geração de regras e em relação à utilização das regras e classes de equivalência geradas. De modo geral, pode-se dizer que a identificação de classes de equivalência obteve êxito para todos os programas. Visto que para os programas *FourBalls* e *GetCmd* todas os casos de teste foram classificados corretamente enquanto para o programa *Triângulo* apenas um caso de teste foi classificado incorretamente e para o programa *Bubble Sort* apenas 7 casos de teste obtiveram classificação incorreta. Quanto à utilização das regras para apoio ao teste de regressão, para o programa *Triângulo*, o conjunto de casos de teste selecionado através da abordagem utilizando PG gerou cobertura equivalente à abordagem *retest-all*, enquanto a abordagem padrão gerou cobertura inferior. Para os demais experimentos, percebe-se que os resultados utilizando a abordagem padrão se mostraram tão bons quanto os resultados utilizando PG. No próximo capítulo é apresentada a conclusão e os trabalhos futuros.

CAPÍTULO 7

CONCLUSÃO

Este trabalho apresentou uma abordagem de aplicação de técnicas de Aprendizado de Máquina para apoio ao teste de regressão, introduzindo uma abordagem genérica para, a partir de diferentes informações resultantes da utilização de ferramentas de teste, identificar as classes de equivalência de um programa, gerar regras envolvendo as três técnicas de teste e através dessas regras, melhorar um conjunto de casos de teste para o teste de regressão.

Foi implementada uma ferramenta chamada RITA que apóia a abordagem. Essa ferramenta possibilita a utilização das técnicas de teste estrutural e baseada em defeitos, e estas, por sua vez, são consideradas complementares por revelarem diferentes tipos de defeitos. A RITA auxilia a aplicação dos critérios dessas técnicas, possibilitando um ambiente integrado, onde a percepção e utilização das relações entre as informações geradas pelas duas técnicas tornou-se possível. Assim, tais relações podem ser utilizadas para apoiar o teste de regressão. As principais limitações da ferramenta implementada tratam da instalação e configuração correta das ferramentas *Poke-Tool* e *Proteum* e do tempo que a ferramenta demora para executar devido à avaliação individual de cada caso de teste em relação ao critério análise de mutantes.

Os resultados obtidos para identificação de classes de equivalência demonstram que, para todos os programas utilizados nos experimentos, os algoritmos de agrupamento conseguiram identificar as classes de equivalência de maneira boa ou satisfatória, ou seja, obtiveram acerto superior à 80%. Não é possível indicar uma técnica como sendo melhor que outra, visto que cada técnica possui características específicas e se adequa a diferentes situações. Esse fato é comprovado ao analisar os melhores resultados obtidos para os programas, onde o algoritmo *K-means* gerou resultados satisfatórios em 3 casos, já o *COBWEB* foi satisfatório em 12 casos, enquanto o *EM* foi satisfatório em 8 casos.

Pôde-se notar que as melhores classificações foram obtidas para os programas *Four-Balls* e *GetCmd*, onde todos os casos de teste foram classificados corretamente quanto às classes de equivalência. Já para os programas *Triângulo* e *Bubble Sort*, alguns casos de teste foram classificados diferentemente da classificação manual.

Já a geração de regras pelo classificador *J48* mostrou-se de grande valor para auxiliar a priorização e redução de um conjunto de casos de teste para o teste de regressão. As regras geradas tomam forma com base nos atributos que as técnicas de agrupamento utilizam para geração dos *clusters*. Essas regras possibilitam a ênfase de um determinado aspecto para o teste de regressão. Sendo assim, as regras possibilitam a seleção de mais casos de teste para as classes de equivalência que obtém melhor desempenho para tal aspecto. Como por exemplo, selecionar mais casos de teste de uma classe de equivalência que possui maior cobertura de um critério estrutural e menos de outra que possui menor cobertura.

Para o programa *Triângulo*, o conjunto de casos de teste selecionado através da abordagem utilizando PG gerou cobertura equivalente à abordagem *retest-all*, enquanto a abordagem padrão gerou resultados piores. Nos demais experimentos, notou-se que os resultados se mostraram satisfatórios, contudo, os resultados utilizando a abordagem padrão, na qual é selecionado um caso de teste de cada classe de equivalência, se mostraram tão bons quanto aos resultados utlizando PG. Os casos nos quais a abordagem padrão obteve resultado tão bom quanto à abordagem utilizando PG são considerados satisfatórios de qualquer maneira, mesmo sem utilizar as regras para obter o melhor resultado.

7.1 Trabalhos Futuros

Este trabalho contribuiu com uma nova aplicação das técnicas de AM. Especialmente as sub-áreas de agrupamento e classificação são envolvidas nessa nova aplicação, a primeira sub-área foi aplicada na identificação de classes de equivalência e a segunda na geração de regras para utilização no teste de regressão. Novos trabalhos e variações das técnicas podem ser realizados através da aplicação de outros algoritmos de Aprendizado de Máquina. Neste trabalho foi utilizado o algoritmo *J48* para realização do aprendizado. Outros al-

goritmos que podem ser utilizados incluem Programação Genética e descoberta de regras (*Apriori*), entre outros. Outras técnicas de agrupamento também podem ser utilizadas para identificação de classes de equivalência.

Além da abordagem com outros algoritmos de AM, a abordagem proposta neste trabalho pode ser aplicada sobre outros atributos das bases de dados. Esses outros atributos podem ser: caminhos percorridos, histórico de detecção de defeitos, entre outros. A escolha de outros atributos para realização do aprendizado pode fornecer resultados ainda mais precisos.

Outros experimentos também podem ser feitos utilizando programas mais complexos para avaliar a abordagem proposta. A abordagem foi implementada pela ferramenta RITA que interage com as ferramentas *Proteum* e *Poketool*, mas ela é genérica e pode ser utilizada com outras ferramentas e contextos de teste. Por exemplo o teste de programas orientados a objetos e a aspectos.

BIBLIOGRAFIA

[1] IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, 10 Dezembro 1990.

[2] H. Agrawal, J. R. Horgan, E. W. Krauser, e S. London. Incremental regression testing. *Proceedings of Conference on Software Maintenance*, páginas 348–357, Setembro de 1993.

[3] C. Apte e S. M. Weiss. Data mining with decision trees and decision rules. *Computer Systems*, 13:197–210, 1997.

[4] S. Bates e S. Horwitz. Incremental program testing using program dependence graphs. *POPL'93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, páginas 384–396, Janeiro de 1993.

[5] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 2 edition, 1990.

[6] P. Benedusi, A. Cimitile, e U. De Carlini. Post-maintenance testing based on path change analysis. *Proceedings of Conference on Software Maintenance*, páginas 352–361, Outubro de 1988.

[7] M. J. A. Berry e G. Linoff. *Data mining techniques: for marketing, sales and customer support*. Wiley Computer Publishing, USA, 1997.

[8] D. Binkley. Reducing the cost of regression testing by semantics guided test case selection. *Proceedings of Conference on Software Maintenance*, páginas 251–260, Outubro de 1995.

[9] J. F. Bowring, J. M. Rehg, e M. J. Harrold. Active learning for automatic classification of software behavior. *Proceedings of ACM International Symposium on Software Testing and Analysis*, páginas 195–205, 2004.

[10] L. Breiman, J. H. Friedman, C. J. Stone, e R. A. Olshen. *Classification and Regression Trees.* Chapman e Hall, 1984.

[11] L. C. Briand, Y. Labiche, e Z. Bawar. Using machine learning to refine black-box test specifications and test suites. páginas 135–144, 2008.

[12] J. G. Carbonell, R. S. Michalski, e T. M. Mitchell. An overview of machine learning. J. G. Carbonell, R. S. Michalski, e T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*. Springer, Berlin, Heidelberg, 1984.

[13] L. A. V. Carvalho. *Data mining: A mineração de dados no Marketing, Medicina, Economia, Engenharia e Administração.* Ed. Érica, São Paulo, SP, Brasil, 2 edition, 2002.

[14] M. L. Chaim. Poketool - uma ferramenta para suporte ao teste estrutural de programas baseados em análise de fluxo de dados. Dissertação de Mestrado, DCA/FEEC/UNICAMP, Campinas, Brasil, Abril de 1991.

[15] M.L. Chaim, J.C. Maldonado, e M. Jino. *Manual de Configuração da POKE-TOOL.* Relatório Técnico, DCA/FEEC/Unicamp - RT/008/91, Campinas - SP, Fevereiro de 1991.

[16] M.L. Chaim, J.C. Maldonado, e M. Jino. *Manual do Usuário da POKE-TOOL.* Technical Report, DCA/FEEC/Unicamp - RT/009-91, Campinas - SP, Fevereiro de 1991.

[17] T. Y. Chen e M. F. Lau. Dividing strategies for the optimization of a test suite. *Information Processing Letters*, 60(3):135–141, 1996.

[18] Y. F. Chen, D. S. Rosenblum, e K. P. Vo. Testtube: A system for selective regression testing. *ICSE '94: Proceedings of the 16th international conference on Software engineering*, páginas 211–222, Maio de 1994.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, e Clifford Stein. *Algoritmos, teoria e prática.* Editora Campus, Rio de Janeiro, 2002.

[20] E. O. Costa, S. R. Vergilio, A. Pozo, e G. A. Souza. Exploring genetic programming and boosting techniques to model software reliability. *International Symposium on Software Reliability Engineering*, 11, 2006.

[21] M. E. Delamaro. Proteum: Um ambiente de teste baseado na análise de mutantes. Dissertação de Mestrado, ICM, USP, São Carlos, Brasil, Outubro de 1993.

[22] M. E. Delamaro. *Mutação de Interface: Um Critério de Adequação Inter-procedimental para o Teste de Integração.* Tese de Doutorado, ICM, USP, São Carlos, Brasil, Junho de 1997.

[23] M. E. Delamaro e J. C. Maldonado. Proteum - a tool for the assesment of test adequacy for C programs. *In Conference on Performability in Computing Systems (PCS 96)*, páginas 79–95, Julho de 1996.

[24] M. E. Delamaro, J. C. Maldonado, e M. Jino. *Introdução ao Teste de Software.* Ed. Campus, 2007.

[25] R. A. DeMillo, R. J. Lipton, e F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–43, Abril de 1978.

[26] D. Driankov, H. Hellendoorn, e M. Reinfrank. *An introduction to fuzzy control.* Springer-Verlag, London, UK, 2 edition, 1996.

[27] S. Elbaum, A. Malishevsky, e G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28:159–182, 2001.

[28] U. Fayyad, G. Piatetsky-Shapiro, e P. Smyth. From data mining to knowledge discovery in databases. *Artificial Inteligence Magazine*, 17:37–54, 1996.

[29] K. F. Fischer, F. Raji, e A. Chruscicki. A methodology for retesting modified software. *In Proceedings of the National Telecommunications Conference B-6-3*, páginas 1–6, 1981.

[30] F.G. Frankl. *The use of Data Flow Information for the Selection and Evaluation of Software Test Data.* Tese de Doutorado, Department of Computer Science, New York University, New York, U.S.A., Outubro de 1987.

[31] P. Fränti e J. Kivijärvi. *Randomised local search algorithm for the clustering problem.* Springer-Verlag, London, 2000.

[32] R. J. Funabashi, A. M. R. Vincenzi, M. E. Delamaro, e J. C. Maldonado. Relatório dos operadores de mutação implementados nas ferramentas proteum e proteum/im. Relatório técnico, ICMC-USP, 2001.

[33] I. Granja. Uma ferramenta de apoio ao teste de regressão. Dissertação de Mestrado, DCA/FEEC/UNICAMP, Campinas, Brasil, Dezembro de 1997.

[34] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, e Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, 2001.

[35] J. Han e M. Kamber. *Data mining - Concepts and Techniques.* Morgan Kaufmann, New York, USA, 2001.

[36] M. J. Harrold, R. Gupta, e M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993.

[37] M. J. Harrold e M. L. Soffa. An incremental approach to unit testing during maintenance. *Proceedings of Conference on Software Maintenance*, páginas 362–367, Outubro de 1988.

[38] J. A. Hartigan. *Clustering algorithms.* Wiley New York, 1975.

[39] J. Hartmann e D. J. Robson. Retest-development of a selective revalidation prototype environment for use in software maintenance. *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences.*, 2:92–101, Janeiro de 1990.

[40] J. Hartmann e D. J. Robson. Techniques for selective revalidation. *IEEE Software*, 7(1):31–36, 1990.

[41] B. Hetzel. *The Complete Guide to Software Testing*. John Wiley & Sons, 2 edition, 1988.

[42] J. Holland. *Adaptation in natural and artificial systems*. MIT Press, 1975.

[43] A. K. Jain, M. N. Murty, e P. J. Flynn. Data clustering: a review. *ACM Computing Survey*, 31(3):264–323, 1999.

[44] J. A. Jones e M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *ICSM'01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, páginas 92, Washington, DC, USA, 2001. IEEE Computer Society.

[45] P. C. Jorgensen. *Software Testing: A Craftsman's Approach*. AUERBACH, 3 edition, 2008.

[46] S. Kartalopulos. *Understanding Neural Networks and Fuzzy Logic: Basic Concepts and Applications*. IEEE Press, 1996.

[47] B. W. Kernighan e P. J. Plauger. *Software Tools in Pascal*. Addison-Wesley, 1981.

[48] A. Koscianski e M. S. Soares. *Qualidade de Software, Aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software*. Novatec Editora Ltda., 2006.

[49] J. Koza. *Genetic Programming: on the programming of computers by means of natural selection*. MIT Press, 1992.

[50] J. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, e G. Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[51] J. Laski e W. Szermer. Identification of program modifications and its applications in software maintenance. *Proceedings of Conference on Software Maintenance*, páginas 282–290, Novembro de 1992.

[52] M. Last, M. Friedman, e A. Kandel. The data mining approach to automated software testing. *KDD'03: Proceedings of the nineth ACM SIGKDD International Conference on Knowledge discovery and data mining*, páginas 388–396, 2003.

[53] H. K. N. Leung e L. White. Insights into testing and regression testing global variables. *Journal of Software Maintenance*, 2(4):209–222, 1990.

[54] H. K. N. Leung e L. J. White. A study of integration testing and software regression at the integration level. *Proceedings of Conference on Software Maintenance*, páginas 290–301, Novembro de 1990.

[55] H. K. N. Leung e L. J. White. A cost model to compare regression test strategies. *Proceedings of Conference on Software Maintenance*, páginas 201–208, Outubro de 1991.

[56] Z. Li, Z. Harman, e R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, 2007.

[57] S. P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28:129–137, 1982.

[58] H. J. Lu, R. Setiono, e H. Liu. Effective data mining using neural networks. *IEEE Transactions on Knowledge and Data Engineering*, 8:957–961, 1996.

[59] X. Ma, B. Sheng, Z. He, e C. Ye. A genetic algorithm for test-suite reduction. *Systems, Man and Cybernetics, 2005 IEEE International Conference on*, 1:133–139, 2005.

[60] J. C. Maldonado. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*. Tese de Doutorado, DCA/FEEC/UNICAMP, Campinas, Brasil, Julho de 1991.

[61] L. A. F. Martimiano. Estudo de técnicas de teste de regressão baseado em mutação seletiva. Dissertação de Mestrado, ICMC, USP, São Carlos, Brasil, Novembro de 1999.

[62] T. M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[63] M. C. Monard, G. E. A. P. A. Batista, S. Kawamoto, e J. B. Pugliesi. Uma introdução ao aprendizado simbólico de máquina por exemplos. Didactical Notes 29, ICMC-USP, São Carlos - SP, 1997.

[64] G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

[65] A. J. Offutt, J. Pan, e J. M. Voas. Procedures for reducing the size of coverage-based test sets. *In Proceedings of the Twelfth International Conference on Testing Computer Software*, páginas 111–123, 1995.

[66] A. K. Onoma, W. T. Tsai, M. Poonawala, e H. Suganuma. Regression testing in an industrial environment. *Commun. ACM*, 41(5):81–86, 1998.

[67] T. J. Ostrand e E. J. Weyuker. Using dataflow analysis for regression testing. *Proceedings of the Sixth Ann. Pacific Northwest Software Quality Conference*, páginas 233–247, Setembro de 1988.

[68] M. Polo, M. Piattini, e I. G. Rodriguez. Decreasing the cost of mutation testing with 2-order mutants. Disponível no site da Universidade de Castilla-La Mancha(2009). http://www.inf-cr.uclm.es/www/mpolo/stvr/.

[69] M. Polo, M. Piattini, e I. G. Rodriguez. Decreasing the cost of mutation testing with second-order mutants. *Software Testing, Verification & Reliability*, 19(2):111–131, 2009.

[70] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 6 edition, 2005.

[71] J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[72] S. Rapps e E. J. Weyuker. Data flow analysis techniques for program test data selection. *In 6th International Conference on Software Engineering*, páginas 272–278, 1982.

[73] S. Rapps e E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, Abril de 1985.

[74] S. O. Rezende. *Sistemas Inteligentes - Fundamentos e Aplicações.* Editora Manole Ltda, 2003.

[75] S. O. Rezende, J. B. Pugliesi, E. A. Melanda, e M. F. Paula. *Sistemas Inteligentes. Fundamentos e aplicações.* Ed. Manole, Barueri, SP, Brasil, 2003.

[76] G Rothermel. *Efficient, Effective Regression Testing Using Safe Test Selection Techniques.* Tese de Doutorado, Clemson University, Clemson, SC, USA, 1996.

[77] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, e X. Qiu. On test suite composition and cost-effective regression testing. *ACM Trans. Softw. Eng. Methodol.*, 13(3):277–331, 2004.

[78] G. Rothermel e M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. *Proceedings of the 1994 Int'l Symposium Software Testing and Analysis*, páginas 169–184, Agosto de 1994.

[79] G. Rothermel e M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[80] G. Rothermel e M. J. Harrold. A safe, efficient regression test selection technique. *ACM IEEE Transactions on Software Engineering Methodology*, 6(2):173–210, 1997.

[81] G. Rothermel, M. J. Harrold, e J. Dedhia. Regression test selection for C++ software. *Software Testing, Verification & Reliability*, 10:2000, 2000.

[82] G. Rothermel, M. J. Harrold, J. Ostrin, e C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. *ICSM'98: Proceedings*

of the International Conference on Software Maintenance, páginas 34, Washington, DC, USA, 1998. IEEE Computer Society.

[83] G. Rothermel, R. J. Untch, e C. Chu. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[84] M. J. Rummel, G. M. Kapfhammer, e A. Thall. Towards the prioritization of regression test suites with data flow information. *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, páginas 1499–1504, 2005.

[85] P. Saraph, M. Last, e A. Kandel. Test case generation and reduction by automated input-output analysis. *IEEE International Conference on Systems, Man and Cybernetics*, 1:768–773, Outubro de 2003.

[86] B. Sherlund e B. Korel. Modification oriented software testing. *In Conference Prigs: Quality Week*, páginas 1–17, Maio de 1991.

[87] A. Srivastava e J. Thiagarajan. Effectively prioritizing tests in development environment. *SIGSOFT Softw. Eng. Notes*, 27(4):97–106, 2002.

[88] A. B. Taha, S. M. Thebaut, e S. S. Liu. An approach to software fault localization and revalidation based on incremental data flow analysis. *Proceedings of the 13th Ann. Int'l Computer Software and Applications Conference*, páginas 527–534, Setembro de 1989.

[89] M. Vanmali, M. Last, e A. Kandel. Using a neural network in the software testing process. *International Journal of Intelligent Systems*, 17(1):45–62, 2002.

[90] F. I. Vokolos e P. G. Frankl. Pythia: a regression test selection tool based on textual differencing. *ENCRESS '97: IFIP TC5 WG5.4 3rd internatinal conference on on Reliability, quality and safety of software-intensive systems*, páginas 3–21, London, UK, UK, 1997. Chapman & Hall, Ltd.

[91] Univeridade Waikato. Weka - machine lerning software in Java. Disponível no site da Universidade de Waikato (2009). http://www.cs.waikato.ac.nz/ml/weka.

[92] S. M. Weiss e N. Indurkhya. *Predictive Data Mining: A Practical Guide*. Morgan Kaufmann, San Francisco, CA, USA, 1998.

[93] I. H. Witten e E. Frank. *Data Mining: practical machine learning tools and techniques with Java implementations*. Morgan Kaufmann Publishers Inc., 2000.

[94] W. E. Wong, J. R. Horgan, S. London, e H. A. Bellcore. A study of effective regression testing in practice. *ISSRE '97: Proceedings of the Eighth International Symposium on Software Reliability Engineering*, páginas 264, Washington, DC, USA, 1997. IEEE Computer Society.

[95] S. S. Yau e Z. Kishimoto. A method for revalidating modified programs in the maintenance phase. *Proc. COMPSAC'87: 21th Ann. Int'l Computer Software and Applications Conf.*, páginas 272–277, Outubro de 1987.

[96] S. Yoo e M. Harman. Pareto efficient multi-objective test case selection. *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, páginas 140–150, New York, NY, USA, 2007. ACM.

APÊNDICE A

OPERADORES DE MUTAÇÃO DA FERRAMENTA *PROTEUM*

Conforme Funabashi et. al [32] a ferramenta *Proteum* conta com 71 operadores para o teste de unidade. O conjunto de operadores de mutação de unidade é classificado em quatro classes: Mutação de Constantes (iniciados pela letra C), Mutação de Comandos (iniciados pela letra S), Mutação de Operadores (iniciados pela letra S) e Mutação de Variáveis (iniciados pela letra V). Para a realização dos experimentos desse trabalho, foram utilizados todos os operadores disponíveis e estes são apresentados abaixo:

- SBRC: Troca os comandos *break* por comandos *continue*
- SBRn: Execução do comando *break* dentro de um laço, forçando o laço a terminar
- SCRB: Troca os comandos *continue* por comandos *break*
- SCRn: Troca um comando *break* aninhado ou um comando *continue* pela função *continue_out_to_level_n(j)*, $2 <= j <= n$
- SDWD: Troca um comando *do-while* por um comando *while*
- SGLR: Troca os rótulos dos comandos do tipo *goto* por todos os outros rótulos que aparecem na mesma função
- SMTC: É projetado para garantir que o laço gere algum efeito na saída do programa
- SMTT: É projetado para garantir que o laço seja executado pelo menos uma vez
- SMVB: Modela defeitos de uso incorreto de final de comandos, ou seja, uso incorreto de (})
- SRSR: Troca cada comando de uma função por todos os comando *return* que existem na mesma função

- SSDL: É projetado para mostrar que cada comando do programa tem um efeito na saída do mesmo. Sendo assim, quando aplicado, ele deleta cada um dos comandos do programa para verificar se afeta a saída
- SSWM: Modela defeitos de formulação de casos em comandos *switch*
- STRI: É projetado para garantir que a condição de cada comando *if* seja avaliado pelo menos uma vez pelo ramo verdadeiro e pelo ramo falso
- STRP: É projetado para revelar código inacessível no programa
- SWDD: Troca um comando *while* por um comando *do-while*
- OAAA: Troca um operador aritmético com atribuições por outro operador aritmético com atribuições
- OAAN: Troca um operador aritmético sem atribuições por outro operador aritmético sem atribuições
- OABA: Troca um operador aritmético com atribuições por operador *bitwise* com atribuições
- OABN: Troca um operador aritmético sem atribuições por um operador *bitwise* sem atribuições
- OAEA: Troca um operador aritmético com atribuições por um operador plano
- OALN: Troca um operador aritmético sem atribuições por um operador lógico
- OARN: Troca um operador aritmético sem atribuições por um operador relacional
- OASA: Troca um operador aritmético com atribuições por um operador de deslocamento com atribuições
- OASN: Troca um operador aritmético sem atribuições por um operador de deslocamento sem atribuições

- OBAA: Troca um operador *bitwise* com atribuições por um operador aritmético com atribuições
- OBAN: Troca um operador *bitwise* sem atribuições por um operador aritmético sem atribuições
- OBBA: Troca um operador *bitwise* com atribuições por outro operador *bitwise* com atribuições
- OBBN: Troca um operador *bitwise* sem atribuições por um outro operador *bitwise* sem atribuições
- OBEA: Troca um operador *bitwise* com atribuições por um operador plano
- OBLN: Troca um operador *bitwise* sem atribuições por um operador lógico
- OBNG: Reverte o contexto das expressões comparando *bitwise*
- OBRN: Troca um operador *bitwise* sem atribuições por um operador relacional
- OBSA: Troca um operador *bitwise* com atribuições por um operador *swift* com atribuições
- OBSN: Troca um operador *bitwise* sem atribuições por um operador de deslocamento sem atribuições
- OCOR: Troca por um operador de *cast* por todos os outros tipos primitivos da linguagem
- OEAA: Troca um operador plano por um operador aritmético com atribuições
- OEBA: Troca um operador plano por um operador *bitwise* com atribuições
- OESA: Troca um operador plano por um operador de deslocamento com atribuições
- Oido: Modela defeitos que surgem com o uso incorreto dos operadores ++ e - -
- OIPM: Revela defeitos no uso incorreto de expressões que envolvam ++, – e operadores de indireção *.

- OCNG: Modela defeitos em comando de seleção e repetição, revertendo essas condições. Utilizado quando não envolve nenhum operador lógico
- OLAN: Troca um operador lógico por um operador aritmético sem atribuições
- OLBN: Troca um operador lógico por um operador *bitwise* sem atribuições
- OLLN: Troca um operador lógico por um operador *bitwise* sem atribuições
- OLNG: Modela defeitos em comandos de seleção e repetição, revertendo essas condições
- OLRN: Troca um operador lógico por um operador relacional
- OLSN: Troca um operador lógico por um operador de deslocamento sem atribuições
- ORAN: Troca um operador relacional por um operador aritmético sem atribuições
- ORBN: Troca um operador relacional por um operador *bitwise* sem atribuições
- ORLN: Troca um operador relacional por um operador lógico
- ORRN: Troca um operador relacional por outro operador relacional
- ORSN: Troca um operador relacional por um operador de deslocamento sem atribuições
- OSAA: Troca um operador de deslocamento com atribuições por um operador aritmético com atribuições
- OSAN: Troca um operador de deslocamento sem atribuições por um operador aritmético sem atribuições
- OSBA: Troca um operador de deslocamento com atribuições por um operador *bitwise* com atribuições
- OSBN: Troca um operador de deslocamento sem atribuições por um operador *bitwise* sem atribuições
- OSEA: Troca um operador de deslocamento com atribuições por um operador plano

- OSLN: Troca um operador de deslocamento sem atribuições por um operador lógico
- OSRN: Troca um operador de deslocamento sem atribuições por um operador relacional
- OSSN: Troca um operador de deslocamento sem atribuições por outro operador de deslocamento sem atribuições
- OSSA: Troca um operador de deslocamento com atribuições por outro operador de deslocamento com atribuições
- Varr: Modela defeitos de uso incorreto de variáveis do tipo *array*
- VDTR: Requer valores negativos, positivos e zero para cada referência escalar
- Vprr: Modela defeitos de uso incorreto de variáveis do tipo ponteiro
- VSCR: São responsáveis pela mutação de referência e componentes de uma estrutura que são substituídos por referências aos demais componentes da mesma estrutura, respeitando os tipos de componentes
- Vsrr: Modela defeitos de uso de uma variável escalar errada
- Vtrr: Modela defeitos de uso incorreto de variáveis do tipo *structure*
- VTWD: Troca referência escalar pelo seu valor sucessor e predecesssor
- Cccr: Troca constantes por todas as constantes do programa
- Ccsr: Troca referências escalares por constantes
- CRCR: Troca cada referência escalar por constantes requeridas dependendo do tipo da referência